

ARBEITSBERICHTE DES INSTITUTS FÜR MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG (INFORMATIK)

FRIEDRICH ALEXANDER UNIVERSITÄT ERLANGEN NÜRNBERG

HERAUSGEBER : H. Billing, W. Händler, U. Herzog,
F. Hofmann, K. Leeb, P. Mertens, H. Müller, G. Nees,
H. Niemann, D. Seitzer, H. J. Schneider, H. Wedekind

SYMPOS/UNIX - Ein Betriebssystem für homogene
Polyprozessorsysteme

Claus-Uwe Linster

DISSERTATION

BAND 14 · NUMMER 3 · ERLANGEN · MAI 1981

1001 #275

ISSN 0344 - 3515

SYMPOS/UNIX - Ein Betriebssystem für
homogene Polyprozessorsysteme

Digitized by the Internet Archive
in 2026

SYMPOS/UNIX⁺ - Ein Betriebssystem für homogene Polyprozessorsysteme

Der Technischen Fakultät
der Universität Erlangen-Nürnberg
zur Erlangung des Grades
DOKTOR-INGENIEUR

vorgelegt
von
Claus-Uwe Linster

Erlangen 1980

⁺ UNIX is a Trademark of Bell Laboratories



Als Dissertation genehmigt
von der Technischen Fakultät
der Universität Erlangen-Nürnberg

Tag der Anmeldung: 10. Dezember 1980

Tag der Promotion: 6. März 1981

Dekan : Prof. Dr. H. Niemann

Berichterstatter : Prof. Dr. F. Hofmann

Prof. Dr. U. Herzog

Karin und Daniel

Herrn Professor Dr. F. Hofmann möchte ich für die Förderung und Betreuung dieser Dissertation und Herrn Professor Dr. U. Herzog für die Übernahme des Zweitgutachtens danken.

Mein Dank gilt auch Herrn Dr. K.-D. Wendler und Herrn Dr. F.X. Wurm für die sorgfältige Durchsicht der Arbeit, sowie Frau E. Roth für die Unterstützung bei der Erstellung des Manuskripts und der sorgfältigen Anfertigung der graphischen Darstellungen.

Zusammenfassung

In dieser Arbeit wird eine Methode zur Konstruktion von Betriebssystemen für Polyprozessorsysteme vorgestellt und exemplarisch am Beispiel 'SYMPOS' angewendet.

Ziel dieser Arbeit ist es aufzuzeigen, daß es mit relativ geringem Aufwand möglich ist, aus einem geeigneten Monoprocessorbetriebssystem ein Polyprocessorbetriebssystem zu entwickeln. Die Vorteile dieser Methode, sowie deren Möglichkeiten auch anspruchsvolle Entwurfsziele erreichen zu können, werden herausgestellt. Weiter werden die Grenzen dieser Vorgehensweise diskutiert.

Über die bloße Erweiterung des Monoprocessorbetriebssystems hinaus wird das Konzept der lokalen Teilsysteme eingeführt und motiviert, sowie die Möglichkeit vorgesehen, die Hardwarestruktur des Systems dem Benutzer für die Lösung seiner Anwenderprobleme direkt nutzbar zu machen.

Abstract

A method of constructing operating systems for polyprocessor machines is presented and applied by way of example to 'SYMPOS'.

The paper demonstrates the possibility of developing a polyprocessor system from a tractable monoprocessor system at relatively small cost. The method's advantages are shown, as are its possibilities of attaining even more demanding design goals. In addition limitations of this approach are also discussed.

Beyond extensions to the monoprocessor system, the design of its local subsystems is introduced and motivated. The system enables the direct use of the hardware structure in solving the user's application problems.

INHALTSVERZEICHNIS

1.0	Einleitung	4
1.1	Motivation	4
1.2	Systemauswahl	6
1.3	Zielsetzung	7
1.4	Übersicht	9
2.0	Struktur des Monoprozessorbetriebssystems UNIX	11
2.1	Überblick	11
2.2	Prozeßverwaltung	13
2.2.1	Prozessorvergabe	15
2.2.2	Prozeßerzeugung, -abwicklung und -synchronisation	16
2.2.3	Arbeitsspeicher und Hintergrund- speicherverwaltung, Ein/Auslagern von Prozessen	17
2.3	Dateiverwaltungssystem	18
2.4	Gerätetreibersystem	21
2.5	Systemdienstaufrufe	22
3.0	Hardwareaufbau des Polyprozessorsystems	24
4.0	Struktur des Polyprozessorsystems	30
4.1	Allgemeines	30
4.2	Prozeßverwaltung	33
4.3	Segmentmanagementsystem	34
4.4	Freispeicherverwaltung	35
4.5	Koordinierungssystem	35
4.6	Nachrichtenkopplungs- und Kommunikationssystem	36
5.0	Komponenten des Polyprozessorsystems	
	Beschreibung - Spezifikation - Implementierung	37
5.1	Vorbemerkungen	37
5.2	Prozeßverwaltung	40
5.2.1	Überblick	40
5.2.1.1	Zustände eines Prozesses - Prozeßklassifikation	40

5.5.2	Koordinierungssystem für Benutzerprozesse	118
5.5.2.1	Spezifikation	119
5.5.2.2	Implementierung	125
5.6	Nachrichtenkopplungs -und Kommunikations-system	131
5.6.1	Nachrichtenkopplungssystem - Hardware	131
5.6.2	Nachrichtenkopplungssystem - Software	132
5.6.2.1	Sendermodul	133
5.6.2.2	Empfängermodul	133
5.6.2.3	Spezifikation	136
5.6.3	Kommunikationssystem	141
5.6.3.1	'signal'-Operation auf Fremd-prozessoren	142
5.6.3.2	Prozeß/Prozessorwechsel	143
5.6.3.3	Bussystemverwaltung	143
5.6.3.4	Lockoperationen ohne Speicherbelastung bei aktivem Warten	144
5.6.3.5	Alarmsystem	145
5.6.3.6	Test- und Überwachungssystem	146
6.0	Betriebssystemschnittstelle, Binder und Systemdienstbibliothek	147
6.1	Betriebssystemschnittstelle	147
6.2	Binder und Systemdienstbibliothek	150
7.0	Benutzersysteme paralleler Prozesse	151
7.1	Ein Beispiel	152
8.0	Schlußbemerkungen	161
	Literaturverzeichnis	

1.0 Einleitung

1.1 Motivation

Betriebssysteme zu entwerfen und zu implementieren sind in der Regel Vorhaben, die einen Aufwand zwischen 20- und 200 Mannjahren erfordern. So gibt es z.B. schon Hersteller in der mittleren Datentechnik, die nicht ohne Stolz auf mehrere 100 Mannjahre Aufwand für ihre Betriebs- und Programmiersysteme (Compiler, Assembler, Binder, Lader, Editoren, etc.) hinweisen.

Die steigende Komplexität der Hardware bei der hier betrachteten Klasse der Poly-(Multi-)prozessorsysteme¹ läßt nicht erwarten, daß die Betriebssysteme weniger komplex werden. Neben den klassischen Betriebssystemaufgaben sind zusätzliche, sich zum Teil widersprechende Forderungen zu erfüllen: z.B. zum einen dem Anwender gegenüber die Komplexität des Gesamtsystems zu mildern und zum anderen die vorhandenen Hardwarestrukturen dem Benutzer zur Lösung seiner Aufgaben voll nutzbar zu machen.

Letztlich wird vom Anwender die 'Güte' eines Systems an der Benutzerschnittstelle gemessen: Für ihn ist die Handhabbarkeit des Systems ebenso wichtig wie die Leistungsfähigkeit.

Unter Benutzerschnittstelle soll hier die Gesamtheit der dem Benutzer zur Verfügung stehenden Software (Programmier-system) verstanden werden. Ein Programmiersystem läßt sich wie folgt gliedern:

- 1) Betriebssystemdienste
- 2) Kommandoprozessor (Kommandosprache)

1.) Homogene total speichergekoppelte und nachrichtengekoppelte Systeme (Def.: Siehe Kap. 3)

- 3) Assemblierer (Assemblersprache)
- 4) Compiler (Höhere Programmiersprachen)
- 5) Laufzeitbibliotheken, Programmbibliotheken
- 6) Dienstleistungsprogramme, wie z.B. Editoren, Textverarbeitungs-systeme, Fehlersuchhilfen, etc.

Bei den bekannten und realisierten Polyprozessorsystemen wie z.B. C.mmp, Cm* und EGPA hat man beim Aufbau der Hardware markt-gängige Prozessoren verwendet, die ursprünglich für eine derartige Nutzung nicht konstruiert worden waren. So ist es bei den genannten Systemen unmöglich, daß ein Prozessor den Prozessorstatus (Registerinhalte, Programmstatuswort, Segmentierungs-(Paging-)Register, etc.) eines anderen Prozessors direkt lesen und schreiben kann. Dies ist jedoch eine wesentliche Voraussetzung, um die Zuverlässigkeit solcher Systeme wirklich steigern zu können.

Als Argument, trotzdem markt-gängige Prozessoren einzusetzen, wird im allgemeinen die Zeitersparnis angeführt und vielleicht auch die Hoffnung, daß die Betriebssystemsoftware Mängel in der Hardwarefähigkeit kompensieren kann, was - betrachtet man den Zuverlässigkeitsaspekt - gewiß nicht immer möglich ist.

Bei der Konstruktion eines Betriebssystems für ein mit markt-gängigen Prozessoren aufgebautes Polyprozessorsystem steht man, wenn die Möglichkeiten, ein derartiges Projekt an einer Hochschule durchzuführen, berücksichtigt werden, vor folgenden Problemen:

- 1) man kann die Eigenschaften der Prozessoren nicht an spezielle Anforderungen einer Polyprozessorstruktur anpassen
- 2) man kann die Voraussetzungen für eine gute Benutzerschnittstelle schaffen. Wegen des außerordentlichen Aufwands ist man aber nicht in der Lage, die in den o.g. Punkten 4-6 aufgeführten Software selbst zu erstellen.

Sinnvoll kann hier nur eine ähnliche Vorgehensweise wie bei der Hardwarekonstruktion sein:

Ein geeignetes Monoprozessorbetriebssystem unter Beibehaltung der Betriebssystemschnittstelle zu einem Betriebssystem für eine Polyprozessoranlage zu erweitern.

Durch die Beibehaltung der Monoprozessor - Betriebssystem-schnittstelle steht dann die gesamte, oberhalb der Betriebssystem-schnittstelle vorhandene Software zur Verfügung. Zusätzlich ist es "nur" noch erforderlich, dem Benutzer die polyprozessorspezifischen Funktionen in geeigneter Weise zur Verfügung zu stellen. Für den 'Monoprozessorbenutzer' hat diese Vorgehensweise den großen Vorteil, auch das Polyprozessorsystem, wie gewohnt, zur Lösung seiner Probleme nutzen zu können.

1.2 Systemauswahl

Durch die Wahl der Prozessoren ist die Auswahl der Betriebssysteme stark eingeschränkt, wenn nicht schon getroffen. Für die der vorliegenden Arbeit zugrundeliegende Hardware (Perkin Elmer (ehemals Interdata)- 7/32-Prozessoren) gibt es erfreulicherweise mehrere verschiedene Betriebssysteme, die auf diesem Prozessortyp ablauffähig sind. Es sind dies die Betriebssysteme:

OS32MT - Version 2	(Perkin Elmer)
OS32MT - Version 3...	(Perkin Elmer)
SOLO	(Brinch Hansen)
UNIX - Version 6	(Western Electric Co.,Inc.)

Die Betriebssysteme OS32MT - Version 2 /Inte76/ und OS32MT - Version 3... /Inte79/ unterscheiden sich im wesentlichen durch verschiedene Speicherverwaltungstechniken (statisch/dynamisch) und durch verschieden mächtige Dateiverwaltungssysteme. Bei dem SOLO-Betriebssystem /Hans77/ handelt

es sich um eine Version, die von der Kansas State University auf die Perkin Elmer Rechner übertragen wurde /Neal77/. Als viertes System steht das UNIX-System Version 6 in einer Perkin Elmer Version der Universität Wollongong, N.S.W. /Mill78/ zur Verfügung.

Bei der Auswahl wurden die Betriebssysteme hinsichtlich folgender Kriterien untersucht:

- 1) Änderungs- und Erweiterungsfreundlichkeit
 - a) modularer Aufbau
 - b) formuliert in einer höheren Programmiersprache
- 2) Vollständigkeit und Qualität der Dokumentation
- 3) Güte und Umfang der Benutzerschnittstelle
- 4) Realisierung eines Mehrprogrammbetriebs (Multiprogramming/ Multitasking)

Dem idealen Betriebssystem am nächsten kam das UNIX-System, welches die Punkte 1b), 3) und 4) voll erfüllte und bei dessen Erstellung der Punkt 1a) nicht völlig außer acht gelassen wurde. Die Perkin Elmer Systeme konnten wegen ihrer ausschließlichen Formulierung in Assemblersprache der Forderung 1) nicht gerecht werden. Das SOLO-System erwies sich vor allem aus zwei Gründen als nicht geeignet: Zum einen stützt sich das hier verfügbare System auf Teile des Perkin-Elmer-Systems OS32MT-Version 2 ab, zum andern ist es als Einbenutzersystem konzipiert.

1.3 Zielsetzungen

Ziel dieser Arbeit ist es, exemplarisch am Beispiel UNIX-Betriebssystem zu zeigen, welche Änderungen und Erweiterungen bei der Entwicklung eines Polyprozessorbetriebssystems aus einem Monoprozessorbetriebssystem erforderlich und welche Funktionsgruppen davon betroffen sind. Die geänderten bzw. neu erstellten Betriebssystemmoduln werden spezifiziert. Es

werden detaillierte Implementierungsvorschläge angegeben, die zum Teil auch realisiert worden sind. Die Implementierung des Gesamtsystems wird auf etwa drei Mannjahre veranschlagt; sie konnte daher nicht Teil dieser Arbeit sein.

Die Leistungsfähigkeit speichergekoppelter Polyprozessorsysteme¹ hängt im wesentlichen davon ab, in wie weit es gelingt, den hardwarebedingten Engpaß "gemeinsamer Speicherbus" zu entlasten. Dieses Bestreben führte schon bei dem Betriebssystem des Projekts C.mmp HYDRA /Wulf74, Wulf75, Jone80/ dazu, den Code besonders häufig angesprochener Betriebssystemfunktionen (z.B. primitive Koordinierungsoperationen) in lokalen Speichern abzulegen. Dieser Weg wird hier noch konsequenter beschritten: Alle lokal abwickelbaren Funktionen und deren Datenstrukturen, sowie der Code der globalen Funktionen wird vorzugsweise in lokalen Speichern abgelegt. Der Speichermehraufwand ist bei den heutigen Preisen für Halbleiterspeicher vertretbar.

Ein derartiges Konzept führt zu einem Betriebssystem, das sich aus einer Anzahl nicht notwendiger Weise identischer lokaler Teilsysteme und einem gemeinsamen Systemkern zusammensetzt. Für die Teilsysteme gelten die Forderungen, die Betriebssystemschnittstelle zum Benutzer und die Schnittstelle zum gemeinsamen Systemkern einzuhalten.

Die Hardwareschnittstelle kann hier durchaus unterschiedlich sein; es ist also denkbar, verschieden mächtige Prozessoren einer Rechnerfamilie (z.B. Perkin Elmer - Prozessoren 8/32, 7/32 und 7/16) zusammenzuschalten. Das Resultat wäre eine heterogene symmetrische (bzw. asymmetrische) speichergekoppelte Polyprozessorkonfiguration¹. Systeme dieser Kategorie bergen jedoch die Gefahr in sich, daß durch die notwendigen Koordinierungsmaßnahmen die schnellen Prozessoren von den langsameren behindert werden /Jone80/. Es

1.) Def.: Siehe Kap. 3

ist also für die hier betrachtete Klasse von Systemen durchaus sinnvoll, eine homogene Hardwarekonfiguration zu fordern.

Der Wunsch, in einem Polyprozessorsystem Teilsystemen spezielle Aufgaben zu übertragen, sollte auch in einem homogenen System möglich sein. Angestrebt wird hier, auf der Basis der Teilsysteme eine Betriebssystemfamilie zu realisieren. Parametrisiert soll es möglich sein, Teilsysteme verschiedener Mächtigkeit (z.B. mit und ohne Ein/Auslagermechanismen) und/oder Systeme verschiedener Fähigkeit (z.B. ausgeprägte 'Timesharing' - Systeme oder ausgeprägte 'Real-time' - Systeme) zu generieren.

Desweiteren sollte es bei einem derartigen Konzept möglich sein, Systeme auf bestimmte Anforderungen hin, denkbar wäre, z.B. ein Prozeßsteuerungssystem mit festliegender Anzahl von Systemprozessen und eindeutigen Beziehungen dieser Prozesse untereinander, eine 'maßgeschneiderte' Schedulingpolitik zur Verfügung zu stellen. Voraussetzung für ein solches Vorhaben ist der streng modulare Aufbau der Systemkomponenten.

Will man den oben aufgestellten Anforderungen genügen, so ist das Ergebnis ein homogenes asymmetrisches speichergekoppeltes Polyprozessorsystem, das den symmetrischen Betrieb als Spezialfall einschließt.

1.4 Übersicht

Im Anschluß an dieses einleitende Kapitel wird die Struktur des dieser Arbeit zu Grunde liegenden Monoprozessorbetriebssystem beschrieben, wobei insbesondere auf die Komponenten eingegangen wird, die später bei der Erweiterung für ein Polyprozessorsystem relevant sind.

Im 3. Kapitel wird ein Klassifikationsschema für Polyprozessorsysteme angegeben und anhand dieses Schemas die vorhandene Hardware eingeordnet. Danach erfolgt die

Beschreibung der Einzelkomponenten des Hardwaresystems.

Das 4. Kapitel gibt einen Überblick über das entwickelte Polyprozessorbetriebssystem, dessen Komponenten im Kapitel 5 näher beschrieben werden. Die Teilabschnitte im Kapitel 5 sind so aufgebaut, daß einer verbalen Beschreibung eine formale Spezifikation in der Spezifikationssprache SPECIAL /Robi77/ und ein Implementierungsvorschlag folgt. Es wird insbesondere auf die polyprozessorspezifischen Teile der einzelnen Moduln eingegangen.

Die Beschreibung und Spezifikation der Betriebssystemschnittstellenerweiterung und der daraus resultierenden Modifikationen des Binders und der Erweiterungen der Systemdienstbibliotheken befindet sich im Kapitel 6.

Mit welchen Systemdiensten und Hilfsmitteln ein Benutzer die Hardwarestrukturen bei der Lösung seiner Probleme nutzen kann, wird in Kapitel 7 beschrieben und durch die Angabe eines Beispiels demonstriert.

Im Schlußkapitel wird neben einer kurzen Zusammenfassung die Problematik der Fehlersuche in parallelen Prozeßsystemen, sowie der beabsichtigte Einsatzbereich des Systems angesprochen.

2.0 Struktur des Monoprozessorbetriebssystems UNIX

2.1 Überblick

In der vorliegenden Arbeit wird von der UNIX-Version der Universität Wollongong, Australien, N.S.W. /Mill78/ ausgegangen, die die Version 6 des UNIX-Betriebssystems für Digital Equipment Corporation PDP-11-Rechner auf Perkin-Elmer-Rechner des Typs 7/32 übertragen hat. Das System ist als ausgeprägtes 'Timesharing'-System konzipiert und zeichnet sich durch eine hohe Benutzerfreundlichkeit aus. Eine große Anzahl der über UNIX erschienenen Artikel sind zusammenfassend im Bell System Technical Journal /Bell78/ veröffentlicht. Im einzelnen werden dort folgende Themen behandelt:

- Einführung und Überblick /Ritc78c/
- Betriebssystemimplementierung /Thom78/
- Eine Betrachtung über die Stärken und Schwächen des Systems, über seine Zuverlässigkeit und Leistungsfähigkeit /Ritc78b/
- Kommandoprozessor und Kommandosprache /Bour78/
- Die Programmiersprache 'C' /Ritc78a/

Einige weitere Artikel befassen sich mit der Portabilität des Systems, sowie mit einigen von UNIX abgeleiteten Spezialsystemen.

Veröffentlichungen, die die Struktur und die Implementierung des UNIX-Systems beschreiben /Thom78, Ritc78c/, gliedern das System in die Systemkomponenten Prozeßverwaltung, Ein/Ausgabesystem und Dateisystem.

In der Prozeßverwaltung werden sämtliche Aktivitäten, die zur Ablaufsteuerung eines Prozesses notwendig sind, zusammengefaßt.

Im Ein/Ausgabesystem wird unterschieden zwischen strukturierter Ein/Ausgabe (block I/O-System) und unstrukturierter Ein/Ausgabe (character I/O-System). Neben den eigentlichen Gerätetreibern sind für beide E/A-Systeme typspezifische Puffermanagementsysteme implementiert. Bei der block-strukturierten Ein/Ausgabe ist mit dem Puffermanagement ein Cachesystem mit 'read ahead / write after'- Mechanismus realisiert. Hierdurch konnte die Häufigkeit der physikali-

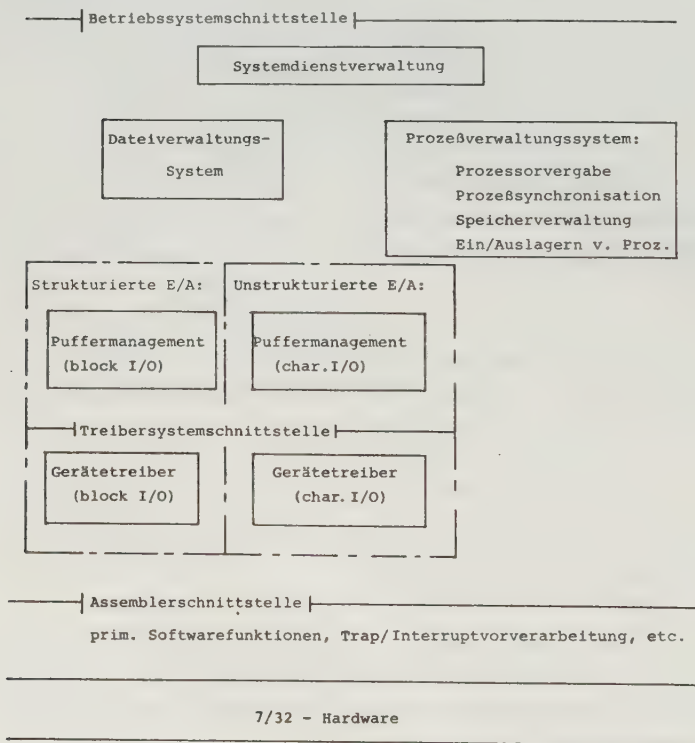


Abb. 2.1 Monoprozessorbetriebssystemstruktur

schen E/A-Transfers reduziert und die Zugriffsgeschwindigkeit im Mittel gesteigert werden. Probleme entstehen bei derartigen Mechanismen hinsichtlich der Datenkonsistenz bei unvorhergesehenen Ereignissen, wie z.B. E/A-Fehler oder (hier sehr seltenen /Ritc78b/) Systemzusammenbrüchen.

Die in der UNIX-Literatur besonders hervorgehobene Systemkomponente ist das Dateisystem, das als sogenannte 'special files' auch die Verwaltung der physikalischen E/A-Geräte organisiert /Thom78/.

Die in Abb. 2.1 dargestellte Struktur gibt die hierarchische Gliederung des nur grob modularisierten Monoprozessorbetriebssystems wieder. Die Moduln sind gemäß ihrer 'logischen' Benutztrelation in Ebenen angeordnet.

Klar definiert sind die Schnittstellen Benutzer/Betriebssystem (Betriebssystemschnittstelle), Betriebssystem-/Peripheriegerätetreibersystem (Treibersystemschnittstelle) und die Schnittstelle des prozessorhardwareabhängigen/-unabhängigen Teils des Systems (Assemblerschnittstelle).

2.2 Prozeßverwaltung

In dieser Systemkomponente sind alle Aktivitäten zur Steuerung des Ablaufs eines Prozesses zusammengefaßt.

Im UNIX - System wird der Begriff Prozeß gemäß der herkömmlichen Definition eines Prozesses verwendet. Die Bezeichnungen 'Benutzerprozeß' und 'Systemprozeß' spezifizieren jedoch nur die Umgebung (Systemumgebung/Benutzerumgebung) in der sich ein Prozeß zu einem bestimmten Zeitpunkt befindet. Da Systemdienste in Form von Unterprogrammen zur Verfügung gestellt werden und Parallelaktivitäten eines Prozesses zu eigenen Ein/Ausgabeoperationen nicht möglich sind, werden mit den obigen Bezeichnungen Phasen eines Prozesses herkömmlicher Definition beschrieben.

Der virtuelle segmentierte Adreßraum eines Prozesses setzt sich aus dem Code(text)segment(reentrant), dem Datensegment und dem Stacksegment zusammen. Aus der Abb. 2.2, die den Zusammenhang zwischen den Prozeßkontrollstrukturen des Betriebssystems und den Prozeßkomponenten aufzeigt, geht hervor, daß mit den Daten- und Stacksegmenten in einem zusammenhängenden Speicherbereich ein prozeßspezifisches System-Datensegment abgelegt ist. Das System-Datensegment, auf das ausschließlich das System Zugriffsrechte besitzt, enthält alle prozeßspezifischen Kontrollinformationen, die nur dann benötigt werden, wenn der Prozeß im Arbeitsspeicher geladen ist. Der Grund für die Zusammenlegung der drei Datensegmente zu einem zusammenhängenden Speicherbereich, ist die einfachere und effizientere Handhabung bei der Ein/Auslagerung eines Prozesses.

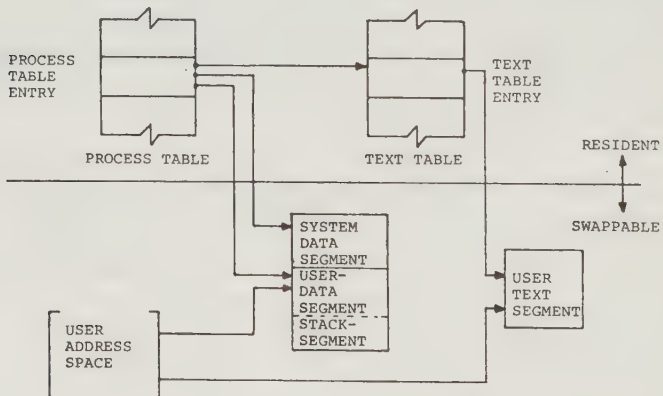


Abb. 2.2 Datenstrukturen der Prozeßverwaltung

Die für die Prozeßverwaltung notwendigen residenten Informationen werden in den Prozeßkontrollblöcken('process table') gehalten. Für die aufwendige Organisation der gemeinsam benutzbaren Segmente existieren eigene Kontrollblöcke, die in der Codesegmenttabelle('text table') zusammengefaßt sind. Die entsprechenden Informationen für die (prozeßspezifischen) Datensegmente sind dagegen direkt in den Prozeßkontrollblöcken abgelegt.

2.2.1 Prozessorvergabe

Jedem Prozeß ist eine Priorität zugeordnet. Systemprozessen wird jeweils dann eine neue Priorität zugeteilt, wenn sie blockiert werden. Abhängig ist diese Priorität von dem Ereignis, das den Prozeß wieder in einen ablauffähigen Zustand setzen wird.

Die Benutzerprozesse werden auf einer niedrigeren Prioritätsebene als die Systemprozesse abgewickelt. Die Priorität ist dabei abhängig von dem Verhältnis Prozessornutzungszeit / reale Zeit eines Prozesses. Die Neuberechnung findet in der Regel in Sekundenabständen statt. Der erwünschte Effekt dieses Prioritätenvergabealgorithmus ist die Bevorzugung interaktiver Prozesse. Prozesse hoher Priorität werden nach intensiver Prozessornutzung auf niedrigerer Prioritätsebenen herabgesetzt, während die Priorität der Prozesse, die auf niedriger Priorität längere Zeit keinen Prozessor zugeteilt bekamen, heraufgesetzt wird. Prozesse gleicher Priorität werden nach 'round robin' (Zeitquantum: 1 Sekunde) abgearbeitet.

Höherpriore Prozesse können unter Berücksichtigung eines Schwellwertes niederpriore Prozesse vom Prozessor verdrängen.

2.2.2 Prozeßerzeugung, -abwicklung und -koordinierung

Das Erzeugen eines Prozesses einerseits und das Laden eines Programms und dessen Abwicklung andererseits sind klar getrennte Aktivitäten (Systemdienste) des Betriebssystems.

Erzeugt wird ein Prozeß mit der Funktion 'fork'. Durch das Anlegen einer Kopie des erzeugenden Prozesses werden die Prozeßverwaltungsstrukturen des neuen Prozesses implizit initialisiert. Nach der Ausführung des Systemdienstes haben beide Prozesse (erzeugender und erzeugter) Zugriff auf dasselbe Codesegment und auf nahezu gleiche Exemplare der Datensegmente. Verschieden sind einige prozeßspezifische Informationen im System-Datensegment und die der Selbstidentifikation der Prozesse dienenden Rückgabeparameter.

Beim Laden eines Ladeobjekts und bei dessen Ausführung ('exec') wird kein neuer Prozeß erzeugt, sondern lediglich der virtuelle Adreßraum des Prozesses verändert. Mindestens das Daten- und das Stacksegment sind von der Modifikation betroffen.

Für die Prozeßkoordinierung sind die Funktionen 'wait' und 'signal' implementiert. Diese Operationen unterscheiden sich von den P- und V-Operationen Dijkstra's /Dijk68/ im wesentlichen dadurch, daß für die Koordinierungsereignisse keine direkten Speicherrepräsentanten (Semaphoren) existieren. Außerdem wird keine explizite Warteschlange von blockierten Prozessen, die auf das gleiche Ereignis warten, geführt.

Hier wird bei der 'wait'-Operation ein Ereigniscode, im allgemeinen die Adresse einer Kontrollstruktur, mit dem das Ereignis auf irgendeine Weise assoziiert ist, angegeben. Das Ereignis "Warten auf Fertigstellung einer Drucker-Ausgabe" wird z.B. mit der Adresse des zugehörigen Gerätekontrollblocks codiert. Dieser "Ereigniscode" wird im Prozeßkontrollblock des aufrufenden Prozesses eingetragen.

Eine 'signal'-Operation hat zur Folge, daß alle Prozesse, die auf das Ereignis warten, in den Zustand "lauffähig" gesetzt werden, ist kein Prozeß vorhanden, so hat die Operation keinen Effekt.

Benutzer des Koordinierungssystems müssen dafür Sorge tragen, daß 'signal'-Operationen nicht vor den dazugehörigen 'wait'-Operationen auftreten können. Dies ist besonders bei hardwaregenerierten 'signal'-Operationen (Interrupts) zu beachten.

2.2.3 Arbeitsspeicher- und Hintergrundspeicherverwaltung, Ein/Auslagern von Prozessen

Die Arbeitsspeicher- und Hintergrundspeicherzuteilung wird nach der Strategie 'First Fit' vorgenommen. Die Verwaltungseinheiten sind beim Arbeitsspeicher 256-Byte-, beim Hintergrundspeicher 512-Byte-Blöcke.

Das Ein/Auslagern von Prozessen (swapping) wird von einem eigenen Systemprozeß organisiert.

Im residenten Teil der Prozeßkontrollblöcke sind u.a. die 'Swapping-Parameter' abgelegt, nach denen die Kandidaten für die Ein- bzw. Auslagerung bestimmt werden. Für die Einlagerung wird derjenige unter den lauffähigen Prozessen bestimmt, der sich die längste Zeit auf dem Hintergrundspeicher befindet. Ist für den einzulagernden Prozeß nicht genügend freier Arbeitsspeicher vorhanden, so wird aus der Menge der blockierten und auf ein 'langsames' Ereignis wartenden Prozesse derjenige ausgewählt, der sich am längsten im Arbeitsspeicher befindet. Um ein zu häufiges Ein/Auslagern zu verhindern, sind für die Auswahlparameter Mindestwerte vorgegeben.

2.3 Dateiverwaltungssystem

In der Abb. 2.3 sind die Datenstrukturen des Dateiverwaltungssystems und ihre Beziehungen untereinander dargestellt.

Pro Benutzerprozeß existiert eine Tabelle der von diesem Prozeß eröffneten Dateien ('per-user open file table'). Um den parallelen Zugriff mehrerer Prozesse auf eine Datei zu ermöglichen, werden die Zeiger auf die nächste zu lesende bzw. zu schreibende Information in einer systemglobalen Tabelle verwaltet ('open file table'). Die dateispezifischen Kontrollinformationen werden ebenfalls systemglobal für alle aktiven Dateien in der Tabelle 'active i-node table' gehalten.

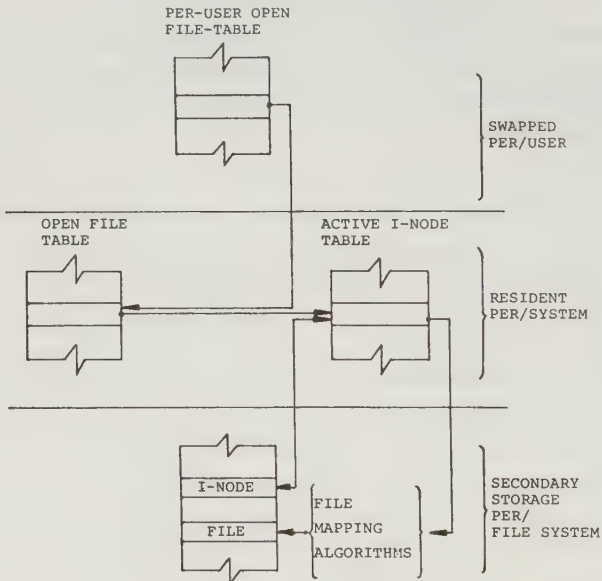


Abb. 2.3 Datenstrukturen des Dateiverwaltungssystems

Für jedes Dateisystem ($\hat{=}$ Plattenlaufwerk) existieren plattenresident ein Beschreibungsblock, der dateisystem-globale Informationen (z.B. freie Speicherbereiche, maximale Anzahl von Dateien, etc.) enthält und eine Liste ('i-list'), deren Elemente Beschreibungen einer Datei sind ('i-node'). Diese Information wird beim Öffnen einer Datei in die Tabelle 'active i-node table' kopiert und beim Schließen einer Datei (aktualisiert) wieder zurückgeschrieben. Die systemeindeutige Identifikation einer Datei wird durch die Kombination des Gerätenamens ('major/minor-number') und der 'i-number' (Index des 'i-node' in der 'i-list') erreicht.

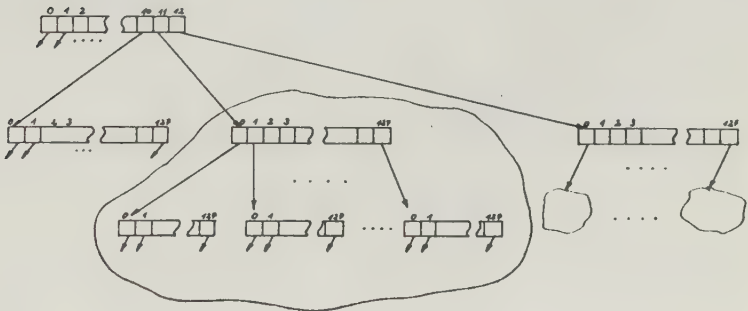


Abb. 2.4 Dateiblockadressierung

In der Struktur 'i-node' befindet sich neben einer Anzahl von Verwaltungsinformationen ein Adreßvektor zur Identifikation der Plattenblöcke (512 Byte) der Datei.

Die ersten 10 Adressen geben direkt die Adressen der ersten 10 Blöcke der Datei an. Ist sie größer als 10 Blöcke, so verweist die 11. Adresse auf einen Block, der die Adressen der nächsten 128 Blöcke der Datei enthält (1x indirekt);

reicht auch diese Größe nicht aus, so wird über die 12. Adresse (2x indirekt) ein Zugriff auf 128^2 Blöcke ermöglicht und schließlich kann über die 13. Adresse (3x indirekt) auf weitere 128^3 Blöcke zugegriffen werden (Abb. 2.4). Der erwünschte Effekt der oben beschriebenen Organisation ist die hohe Zugriffsgeschwindigkeit bei kurzen Dateien (≤ 10 Plattenblöcke). Nach Aussagen von D.M.Ritchie /Ritc78b/ gehören bei Programmentwicklungssystemen etwa 80% der Dateien dieser Kategorie an.

Der bisher beschriebenen physikalischen ist eine logische Organisation überlagert, durch die ein hierarchisches Dateisystem implementiert ist. Durch die Einführung eines speziellen Dateityps 'directory' wird die Abbildung Dateiname --> Datei realisiert. Die Verwaltung solcher Dateien unterscheidet sich nicht von der Verwaltung gewöhnlicher Dateien, lediglich der Zugriff bleibt ausschließlich Systemfunktionen vorbehalten.

Die Wurzel des gesamten Dateisystems (root-directory), residiert auf einem fest zugeordneten Gerät auf einem bestimmten Platz (Systemgenerierparameter).

Mit den jeder Datei zugeordneten Schutzschlüsseln können Teilsysteme realisiert werden, auf die z.B. nur ein Benutzer oder eine Benutzergruppe bestimmte Zugriffsrechte ausüben darf. Das im Dateiverwaltungssystem verankerte Schutzsystem gestattet die Vergabe der Zugriffsrechte lesen ('read'), schreiben ('write') und laden-und-ausführen ('exec'). Die Zugriffsrechte können dem Besitzer (implizit), einer Gruppe von Benutzern, wobei der Besitzer Mitglied dieser Gruppe sein muß, oder allen Benutzern gewährt werden. Nur der Besitzer oder ein ausgezeichnete Benutzer ('super user') können Rechte weitergeben oder verändern. Handelt es sich bei der Datei um eine des Typs 'directory', so bedeutet das Recht 'exec' suchen erlaubt, das Recht 'write' Datei anlegen

oder löschen erlaubt und das Recht 'read' lesen erlaubt.

In Abb. 2.5 wird ein Beispiel für ein mögliches Dateisystem gegeben, eine Zugriffsberechtigung ist aus der gezeigten Struktur nicht ableitbar.

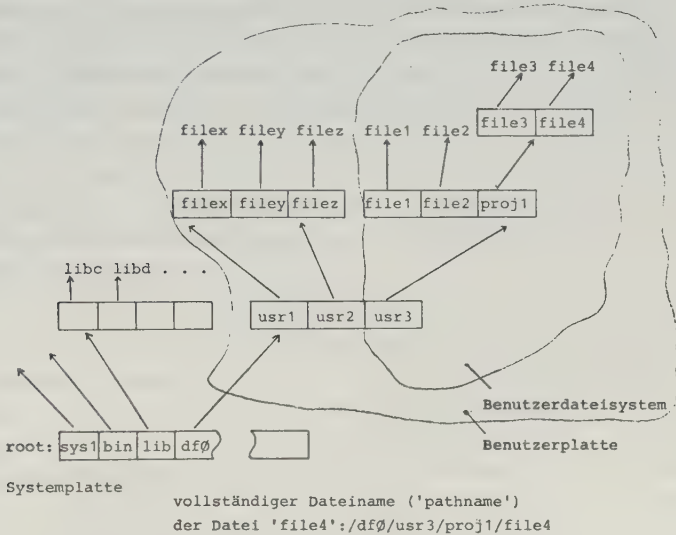


Abb. 2.5 Beispiel für ein hierarchisches Dateisystem residierend auf einer Systemplatte und einer Benutzerplatte, auf der sich drei benutzereigene Dateisysteme befinden.

2.4 Gerätetreibersystem

Wie schon im Überblick erwähnt, gliedert sich das E/A-System in zwei disjunkte Teilsysteme:

der 'strukturierten' Ein/Ausgabe (block I/O-System) und

der 'unstrukturierten' Ein/Ausgabe (character I/O-System).

Ein Gerät wird charakterisiert durch das Paar ('major device number'/'minor device number') und durch seine Zugehörigkeit zu einem der E/A-Systeme.

Für jedes E/A-System existiert als eindeutige Schnittstelle zwischen Treibersystem und Betriebssystem die Konfigurationsmatrix ('configuration table'). In dieser Tabelle befinden sich die Adressen der standardisierten Einsprünge in die einzelnen Treiberfunktionen. Für jeden Gerätetyp ist eine der 'major device number' entsprechende Zeile in der Matrix reserviert. Die Spalten der Matrix geben die Funktionen an, die für die entsprechenden Geräte ausführbar sind. Durch die 'minor device number' wird der Gerätekontrollblock und damit das physikalische Gerät bei mehreren Geräten des gleichen Typs identifiziert.

Der Interrupthandler ist, da er stark maschinenabhängig ist, in Assemblersprache formuliert. Über die Matrix 'devmap' ermittelt er die Einsprungsadresse für die 'Interrupt Service Routine' des betreffenden Geräts, die wie auch die andern Funktionen der Gerätetreiber in 'C' abgefaßt sind. Die primitiven E/A- Funktionen (write byte, read byte, sense status, output command, etc.) sind in Assemblersprache codierte Unterprogramme, die über Standardunterprogrammaufrufe in 'C' von den Treiberfunktionen erreichbar sind.

2.5 Systemdienstaufrufe

Üblicherweise wird bei Systemdienstaufrufen die Schutzumgebung gewechselt. Hierdurch ist gewährleistet, daß sich ein fehlerhaftes Verhalten eines Benutzerprogramms nicht auf Betriebssystemteile auswirken kann.

Bei dem vorliegenden System wird ein Systemdienstaufruf über einen speziellen Maschinenbefehl (SVC) mit einem Systemdienstcode als Parameter aufgerufen.

Wird dieser Befehl ausgeführt, so wird ein 'Trap' verursacht, der die Hardware veranlaßt, einen Programmstatuswortwechsel vorzunehmen, sowie die Registersätze umzuschalten.

Nach einer Trapvorverarbeitung wird in den 'Traphandler' verzweigt, der falls es sich um einen Systemdienstaufruf handelt, den Systemdienstcode als Index in der Systemdienstmatrix ('system entry table') verwendet. In der Systemdienstmatrix - der Schnittstellendefinition Benutzer/Betriebssystem - ist für jeden Systemdienst eine Zeile vorgesehen in der Parameteranzahl und Systemeinsprungsadresse abgelegt sind. Ist der Aufruf korrekt (Existenz des Betriebssystemdienstes, Anzahl der Parameter), so wird zum gewünschten Systemdienst verzweigt.

Je nach Art des Systemdienstes werden dort weitere Prüfungen wie z.B. zulässige Wertebereiche der Parameter, Aufrufberechtigung des Aufrufers abgeprüft, bevor endgültig die gewünschte Funktion ausgeführt wird. Welche Systemdienste im einzelnen zur Verfügung stehen kann den UNIX - Handbüchern /Thom75/ entnommen werden.

3.0 Hardwareaufbau des Polyprozessorsystems

Mit Hilfe eines Klassifikationsschemas für Polyprozessorsysteme /Wend78/ soll zunächst die vorhandene Rechenanlage charakterisiert werden. Dies soll u.a. auch der besseren Vergleichbarkeit mit anderen existierenden Polyprozessorsystemen dienen. In Anlehnung an die oben zitierte Arbeit gelten folgende Definitionen:

Definition 3.1:

Ein Prozessorelement besteht aus einem Prozessor und ggf. einem lokalen Speicher.

Definition 3.2:

Ein Polyprozessorsystem ist ein Rechensystem, welches aus mindestens zwei Prozessorelementen besteht. Alle Prozessorelemente stehen unter der Kontrolle eines einzigen Betriebssystems.

Definition 3.3:

- 1) Ein Polyprozessorsystem heißt speichergekoppelt, wenn mindestens zwei Prozessorelemente auf einen gemeinsamen Speicher zugreifen können.
- 2) Ein Polyprozessorsystem heißt total speichergekoppelt, wenn alle Prozessorelemente auf einen gemeinsamen Speicher zugreifen können.
- 3) Ein Polyprozessorsystem heißt nachrichtengekoppelt, wenn alle Prozessorelemente durch ein Nachrichtenverbindungsnetz miteinander kommunizieren können.

Definition 3.4:

In einem symmetrischen Polyprozessorsystem ist jeder Prozessor befähigt, alle im Rahmen des Betriebsablaufs anfallenden Aufgaben auszuführen.

Definition 3.5:

In einem asymmetrischen Polyprozessorsystem sind nicht alle Prozessoren zu allen im Rahmen des Betriebsablaufs anfallenden Aufgaben befähigt.

Definition 3.6:

Ein homogenes Polyprozessorsystem enthält nur Prozessorelemente eines Typs.

Definition 3.7:

Ein heterogenes Polyprozessorsystem enthält Prozessorelemente unterschiedlichen Typs.

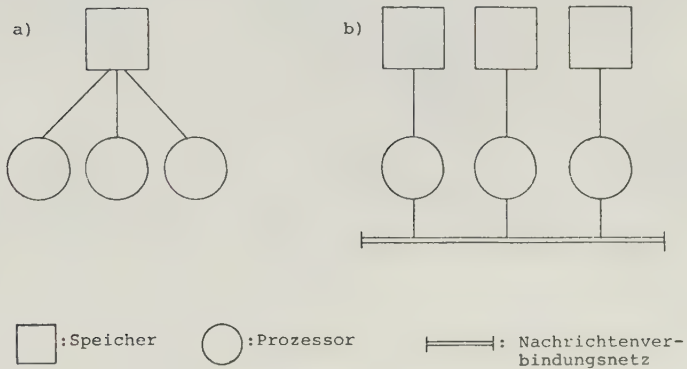


Abb. 3.1 Polyprozessorsystem a) total speichergekoppelt
b) nachrichtengekoppelt

Das in Abb. 3.2 dargestellte Perkin Elmer 7/32-Rechensystem läßt sich aus Hardwaresicht leicht in das oben angegebene Klassifikationsschema einordnen:

Es ist ein total speichergekoppeltes und nachrichtengekoppeltes homogenes Polyprozessorsystem.

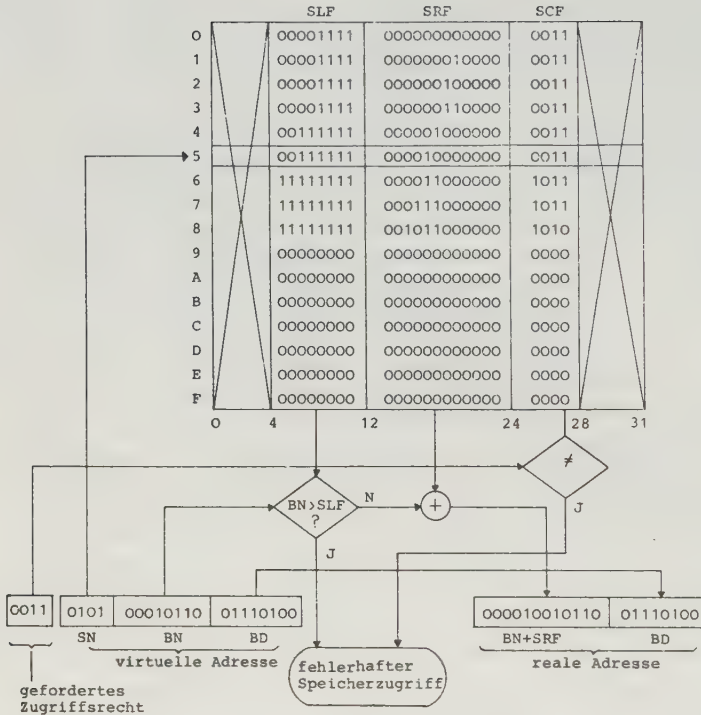
Die Zentraleinheiten (CPU) sind Perkin Elmer Prozessoren des Typs 7/32. Jeder Prozessor verfügt über zwei Registersätze zu je sechzehn 32 Bit breiten Registern (General Purpose Register).

Der Zugriff auf das Arbeitsspeichersystem erfolgt über eine Segmenttabelle, die aus 16 Segmentierungsregistern besteht. Ein Register setzt sich aus einem Grenzregister (Segment Limit Field), einem Basisregister (Segment Relocation Field) und einem Zugriffsschutzregister (Segment Control Field) zusammen. Ein weiteres Register dient der Fehlerursachenerkennung bei fehlerhaften Speicherzugriffen. Mit dieser im 'Memory Access Controller' (MAC) realisierten Hardware sind die Voraussetzungen für die Implementierung eines virtuellen segmentierten Adreßraums gegeben. Der physikalische und virtuelle Adreßraum eines Prozessors umfaßt ein Megabyte. Den Mechanismus der einstufigen Adreßumsetzung zeigt Abb. 3.3.

Das Speichersystem gliedert sich in den gemeinsamen Speicher (CM) und in die den einzelnen Prozessoren zugeordneten lokalen Speichern (LM). Durch die Hardware bedingt (Interrupt Service Pointer Table, Mem. Access Controller, etc.) müssen die lokalen Speicher bei der Adresse x'0' beginnen. Die niedrigste physikalische Adresse des gemeinsamen Speichers liegt auf einer Bankgrenze (x'40000', x'80000' oder x'C0000').

Der gemeinsame Speicher ist so geschaltet, daß er von allen Prozessoren unter den gleichen physikalischen Adressen angesprochen wird. Die Zeit eines Zugriffs auf den gemeinsamen Speicher ist bei Konfliktfreiheit etwa um 50% größer als die Zugriffszeit auf den lokalen Speicher. Im Konfliktfall werden die Zugriffe nach der Strategie 'round robin' abgearbeitet. Ein voll paralleler Zugriff mehrerer Prozessoren auf verschiedene Speicherblöcke ist immer möglich, dies gilt auch für verschiedene Bänke des gemeinsamen Speichers. Aus Kostengründen ist bei der vorhandenen Anlage jedoch nur eine Speicherbank im gemeinsamen Speicher vorhanden.

Segmenttabelle



SLF: Segment Length Field
 SRF: Segment Relocation Field
 SCF: Segment Control Field

Abb. 3.3 Adreßtransformation und Zugriffskontrolle

Neben der Speicherkopplung existiert eine Nachrichtenkopplung, die über die E/A-Schnittstelle realisiert ist. Mit dem ICIMM ('Inter-CPU-Interrupt- and Message-Module') ist es möglich, direkt Nachrichten zwischen den Zentraleinheiten auszutauschen ohne über einen 'Polling-Mechanismus' den gemeinsamen Speicher belasten zu müssen. Außerdem unter-

stützt diese Kopplungsebene Mechanismen, die der Überprüfung der Funktionstüchtigkeit einzelner Hardwarekomponenten dienen.

Bei den Peripheriegeräten wird zwischen lokalen und globalen Geräten unterschieden. Die globalen Peripheriegeräte sind über Busschalter (BS) jedem Prozessor zugänglich. Für die Geräte mittlerer und langsamer Geschwindigkeit existiert ein gemeinsamer Multiplexorbus (CMUXBUS), während für die schnellen Geräte (Platten) der gemeinsame Selektorbus (CSELBUS) installiert ist.

4.0 Struktur des Polyprozessorbetriebssystems SYMPOS¹/UNIX

4.1 Allgemeines

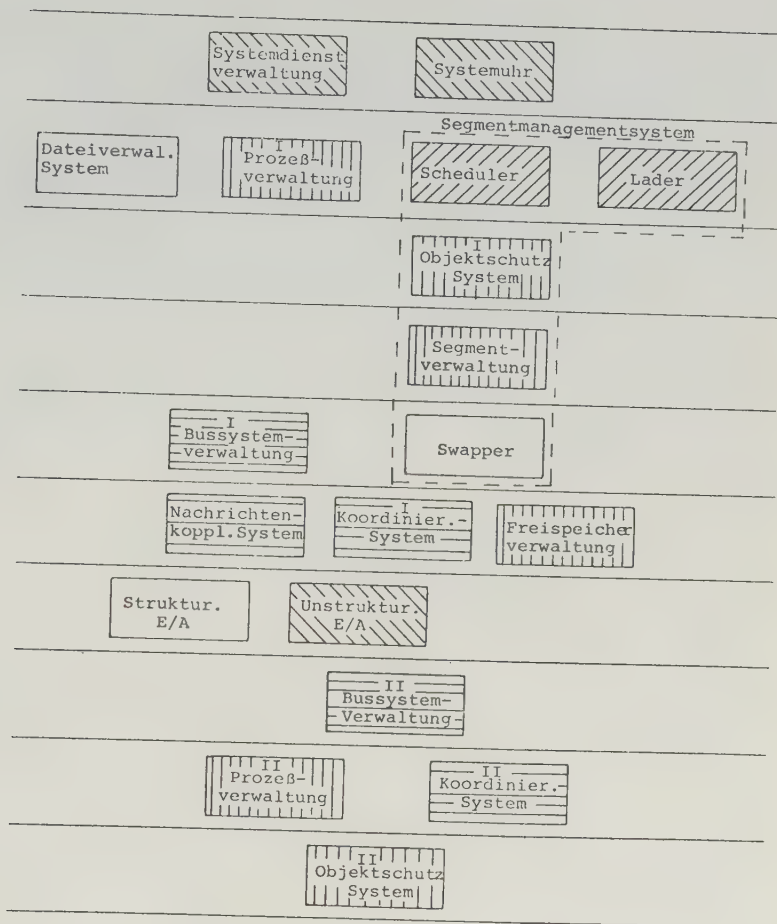
In diesem Kapitel wird ein Überblick über das Polyprozessorbetriebssystem gegeben und erläutert, welche grundsätzlichen Probleme bei der Erweiterung von Monoprozessorbetriebssystemen für Polyprozessorsysteme bei bestimmten gegebenen Voraussetzungen zu lösen sind. Eine detaillierte Beschreibung der von den Modifikationen betroffenen Moduln folgt im Kapitel 5.

Die Abb. 4.1 zeigt die Polyprozessorbetriebssystemstruktur, wobei die Moduln hierarchisch gemäß der Benutzrelation angeordnet sind. Dargestellt ist hier die in der Modulhierarchie höchste Ebene (also die makroskopischste Auflösung), auf der die einzelnen Moduln Funktionsgruppen repräsentieren, die auch von den klassischen Betriebssystemen her bekannt sind. Dieses hohe Modulniveau und die hierarchische Gliederung nach der Benutzrelation führt dazu, daß einige Moduln in der Abb. 4.1 in Einzelkomponenten (römische Ziffern) zerlegt werden mußten.

Der Vergleich mit der Abb. 2.1 zeigt, daß das Polyprozessorbetriebssystem feiner modularisiert ist. Dies ergibt sich aus der Notwendigkeit, das Monoprozessorbetriebssystem im Hinblick auf polyprozessorspezifische Eigenschaften zu untergliedern. Die so gewonnenen Funktionsgruppen lassen sich wie folgt klassifizieren:

1) SYMPOS: SYMmetrical Polyprocessor Operating System

(Das Adjektiv 'symmetrisch' bezieht sich hier auf die homogene Hardware des Systems. Aus historischen Gründen soll jedoch der Name des Betriebssystems nicht geändert werden.)



- ≡ neue Moduln
- \\ geringe Modifikation
- || erhebliche Erweiterungen und Änderungen
- /// wesentliche Erweiterungen

Abb. 4.1 Polyprozessorbetriebssystemstruktur

- Funktionsgruppen, die von der Erweiterung auf ein Polyprozessorbetriebssystem nicht berührt werden.
- Funktionsgruppen, bei denen nur geringe Modifikationen notwendig sind. Hierbei handelt es sich in der Regel um solche, die hinsichtlich des Polyprozessorbetriebssystems lokalen Charakter haben.
- Funktionsgruppen, die stark von der Erweiterung auf ein Polyprozessorbetriebssystem betroffen sind.

Der erste Schritt ist also eine für das Polyprozessorbetriebssystem adäquate Modularisierung des Monoprozessorbetriebssystems. Der Aufwand der Modularisierung ist umso größer, je schlechter das Monoprozessorbetriebssystem strukturiert ist.

Nach der hinreichenden Modularisierung des Monoprozessorbetriebssystems stellt sich die Frage, wo die Modifikationen anzubringen sind und in welchem Umfang das System Änderungen unterworfen werden muß.

Allgemein ist folgendes festzustellen: je heterogener die Hardware und je asymmetrischer das Polyprozessorbetriebssystem konzipiert ist, desto aufwendiger wird die Betriebssystemadaption nach dem hier vorgeschlagenen Verfahren. Ab einem bestimmten Maß an Heterogenität und Asymmetrie wird sich eine Neukonzeption des Betriebssystems mit dem in Kap.1 geschilderten zusätzlichen Aufwand nicht vermeiden lassen.

Am einfachsten ist der Fall eines homogenen symmetrischen speichergekoppelten Polyprozessorsystems ohne lokalen Speichern mit der zusätzlichen Einschränkung, die Schnittstelle zum Benutzer nicht zu verändern. Im wesentlichen beschränken sich in diesem Fall die Änderungen auf folgende Maßnahmen/Srod78/:

- 1) Ergänze alle Daten, auf die von mehreren Prozessen aus zugegriffen werden kann, durch eine Lockvariable.

- 2) Ergänze den Code des Betriebssystems mit 'Test and Set' -Sequenzen dort, wo ein gegenseitiger Ausschluß beim Zugriff auf gemeinsame Daten notwendig ist.

Durch die Verwendung lokaler Speicher, durch die Realisierung eines Nachrichtenkopplungssystems, durch das Konzept lokaler Teilsysteme, sowie durch die Zielsetzung, die Hardwareparallelität dem Benutzer zugänglich und direkt nutzbar zu machen, werden die Systemanforderungen wesentlich verschärft. Die Lösung wird zwangsweise beträchtlich komplexer als im oben beschriebenen einfachsten Fall. Trotzdem erscheint die hier vorgeschlagene Vorgehensweise sinnvoll und effizient. Drei Mannjahre Implementierungsaufwand ist für ein solches System sicher gering.

Abb. 4.1 vermittelt den Eindruck, daß bis auf einige Ausnahmen das gesamte Betriebssystem von den Änderungen betroffen ist. Dies entspricht nicht den Tatsachen. Es ist vielmehr zu berücksichtigen, daß eine gezielte Modularisierung genau an den Stellen des ursprünglichen Betriebssystems vorgenommen wurde, die an die Polyprozessorstruktur angepaßt werden mußten. Einer Vielzahl modifizierter, aber im Codeumfang kleiner Moduln steht somit eine geringe Anzahl unmodifizierter aber sehr umfangreicher Moduln gegenüber.

Die meisten vorhandenen 'Verwaltungsmechanismen' wie z.B. Pufferverwaltung, Verwaltung der verschiedenen Kontrollstrukturen, etc.) konnten auch in die angepaßten Moduln übernommen werden. Außerdem sind die umfangreichen Moduln Betriebssystemdienstverwaltung, E/A-System und der diffizile, in Assemblersprache abgefaßte Systemteil (Kap. 2.) nur geringfügig erweitert worden.

Im folgenden wird ein kurzer Überblick über die notwendigen Modifikationen und Erweiterungen - abgeleitet aus den Zielsetzungen und den polyprozessorspezifischen Gegebenheiten - gegeben.

4.2 Prozeßverwaltung

Da ein Entwurfsziel das der lokalen Teilsysteme ist - resultierend aus der Forderung, lokale Funktionen auch lokal abzuwickeln - erwies es sich als sinnvoll, Prozesse nach ihren lokalen oder globalen Eigenschaften, d.h. nach der Menge der möglichen Zustände im Zustandsraum der Prozesse, zu klassifizieren. So wird bei der Prozeßverwaltung zwischen lokalen, global/temporär lokalen und globalen Prozessen unterschieden.

In dem Modul Prozeßverwaltung sind im Gegensatz zum Modul gleichen Namens im Monoprozessorbetriebssystem, ausschließlich die Funktionen implementiert, die die Zustandsübergänge der Prozesse und implizit auch die der Prozessoren realisieren. Wenn man sich die Betrachtungsweise von Denning /Denn76/ zu eigen macht, so werden hier die virtuellen Prozessoren (die Prozesse) auf die realen abgebildet.

Sowohl für die Prozessorvergabe- als auch für die Lastverteilungsstrategie (Systembalance) sind getrennte interne Moduln vorgesehen.

4.3 Segmentmanagementsystem

Die Entscheidung, wo ein Segment im Arbeitsspeicher- und Hintergrundspeicher plaziert wird, ist bei Polyprozessorsystemen nicht mehr ein isoliertes Problem der Freispeicherverwaltung, sondern ist zusätzlich abhängig von der Klassenzugehörigkeit des Prozesses an den ein Segment gebunden ist und von der physikalischen Verbindungsstruktur des Systems. Die bisherigen Unterscheidungsmerkmale der Segmenttypen mußten hinsichtlich der Plazierungseigenschaften der Segmente erweitert werden.

Im Segmentmanagementsystem, welches die Moduln 'Scheduler', Lader, Segmentverwaltung, Objektschutzsystem und 'Swapper'

umfaßt, waren von dem oben beschriebenen Problem vor allem die Moduln Lader, Segmentverwaltung und Objektschutzsystem betroffen, im geringeren Umfang der Strategiemodul der Arbeitsspeicher/Hintergrundspeicherverwaltung ('Scheduler'). Von den Änderungen nicht berührt war der 'Swapper', da er nur für den physikalischen Transfer zwischen Arbeitsspeicher und Hintergrundspeicher verantwortlich ist.

4.4 Freispeicherverwaltung

Die Änderungen bzw. Erweiterungen in der Freispeicherverwaltung waren notwendig, um folgenden Forderungen gerecht werden zu können:

1. Von der Speicherverwaltung ist zu berücksichtigen, daß das Polyprozessorbetriebssystem im Gegensatz zum Monoprozessorbetriebssystem keinen zusammenhängenden Speicherbereich belegt (lokale Teilsysteme/gem. Systemkern).
2. Durch die hardwarebedingte grobe Rasterung des Arbeitsspeichers in Arbeitsspeicherbänke zu 256K Byte ist es sinnvoll, einen möglicherweise lückenhaften Adreßraum während der Initialisierungsphase des Systems zu erkennen und bei der Verwaltung berücksichtigen zu können. Hierdurch ist es nicht notwendig die lokalen Speicher voll auszubauen. Außerdem werden defekte Speichereinheiten erkannt und der verfügbare Speicher auf der Konsole protokolliert.
3. Die Freispeicherverwaltung muß getrennt für die Speicherklassen lokaler Arbeitsspeicher, gemeinsamer Arbeitsspeicher und Hintergrundspeicher erfolgen.

4.5 Koordinierungssystem

Da die vorhandenen Koordinierungsmechanismen nicht den Anforderungen eines Polyprozessorsystems genügen (Kap. 2.2.2), war hier eine Neukonzeption notwendig. Im Koordinierungssystem sind die Moduln der Koordinierungsmechanismen zusammengefaßt. Implementiert sind ein Lock/Unlock-Mechanismus (binärer Semaphor), ein PV-System, das ausschließlich vom Betriebssystem selbst genutzt wird, und ein PV_{chunk}-System, das dem Benutzer zur Verfügung steht.

4.6 Nachrichtenkopplungs- und Kommunikationssystem

Aufgrund der Überlegungen, den gemeinsamen Speicher als potentiellen Engpass zu entlasten, sollte es vermieden werden, notwendige systemglobale Koordinierungsmaßnahmen durch 'Polling'-Mechanismen realisieren zu müssen.

Hierfür wurde das Nachrichtenkopplungssystem installiert, welches den direkten Austausch von Nachrichten zwischen den Teilsystemen mit Hilfe eines Interruptmechanismus gestattet. Realisiert wird die Nachrichtenvermittlung über Nachrichtenprozesse, die Funktionen des Nachrichtenmoduls und Funktionen der das Nachrichtenkopplungssystem benutzenden Systemmoduln aufrufen, wobei auf eine klare Aufgabentrennung Wert gelegt wird. In der Verantwortung des Nachrichtenmoduls liegt das Senden und Empfangen von Nachrichten, die Zuordnung der Nachricht zum Empfänger und die Koordinierung des Nachrichtenprozeßsystems; in der Verantwortung der einzelnen Moduln liegt der Effekt, den eine übermittelte Nachricht auslöst, sowie der Inhalt einer zu sendenden Nachricht. Das Nachrichtenkopplungssystem und die Modulfunktionen, die von den Nachrichtenprozessen aufgerufen werden, werden in der vorliegenden Arbeit zusammenfassend als Kommunikationssystem bezeichnet.

5.0 Komponenten des Polyprozessorbetriebssystems Beschreibung, Spezifikation, Implementierung

5.1 Vorbemerkungen

Zur Spezifikation der einzelnen Moduln wurde die Spezifikationssprache "SPECIAL" /Robi77/ herangezogen. Abweichungen wurden nur dann erwogen, wenn dadurch Sachverhalte klarer herausgestellt werden konnten. Aus Gründen der Übersichtlichkeit und der kürzeren Schreibweise wurden für einige Operatoren, Quantoren, etc. die gängigen mathematischen Symbole eingeführt.

Dies gilt für:

NOT	≐	$\neg$
AND	≐	$\wedge$
OR	≐	$\vee$
INSET	≐	$\in$
NOT INSET	≐	$\neg \in$
EXISTS	≐	$\exists$
NOT EXISTS	≐	$\neg \exists$
FORALL	≐	$\forall$
EMPTY SET	≐	$\emptyset$
CARDINALITY(x)	≐	$ x $
DIFF	≐	$\setminus$

Weiterhin gilt:

- EXTRACTOR ≐ Verweis auf ein Element einer Menge
- $abcd_i$ ≐ Bezeichnung eines Elements einer Menge oder Bezeichnung einer Teilmenge

Zur Koordinierung

Obwohl es bei Spezifikationssprachen Ansätze gibt, die es gestatten das Problem der Verträglichkeit paralleler Prozesse und deren Koordinierung zu formulieren /Kera79/, wurde auf eine Verwendung in dieser Arbeit verzichtet. Die entwickelten Konzepte, die für alle Moduln gleich sind, sollen hier zusammenfassend erläutert werden:

1. Objekte, auf die ein paralleler Zugriff möglich ist, werden in zwei Klassen eingeteilt:
 - a.) Objekte, die einen Zugriff mit Interruptsperre gestatten. (kurze kritische Abschnitte)
Koordinierung: lock/unlock (Kap. 5.5.1.1)
 - b.) Objekte, die keinen Zugriff mit Interruptsperre gestatten. (lange kritische Abschnitte)
Koordinierung: syssem (Kap. 5.5.1.2)
2. Objekte gleichen Typs (z.B. Prozeßkontrollblöcke, Segmentkontrollblöcke, etc.) werden zu Vektoren zusammengefaßt, um echt parallele Suchvorgänge ohne gegenseitigen Ausschluß gestatten zu können.
3. Prozesse, die manipulierend auf Daten zugreifen, müssen innerhalb des kritischen Abschnitts als erstes die 'Suchbedingung' überprüfen und ggf. den kritischen Abschnitt ohne Manipulation der Daten wieder verlassen.

Vorteile:

- Suchvorgänge sind grundsätzlich echt parallel möglich; - bei verketteten Listen nur durch Teilung der Listen und wesentlich erhöhten Listenverwaltungsaufwand realisierbar
- Systemdatenstrukturen sind invariant gegenüber der Anzahl der Prozessoren

- Keine dynamische Speicherverwaltung im Betriebssystembereich
- Geringer dynamischer Verwaltungsaufwand

Nachteile:

- Nicht genutzte Speicherbereiche

Zusammenfassend:

Verbesserung des dynamischen Verhaltens wird mit vergleichsweise geringer 'Speicherverschwendung' erkaufte.

5.2 Prozeßverwaltung

5.2.1 Überblick

5.2.1.1 Zustände eines Prozesses - Prozeßklassifikation

Das in der Abb. 5.2.1 dargestellte Zustandsdiagramm beschreibt die möglichen Zustände eines Prozesses im Mono-prozessorsystem. Um das Zustandsdiagramm vollständig für ein Polyprozessorsystem zu entwerfen, wäre es notwendig, dieses Diagramm in n Ebenen (n = Anzahl der Prozessoren) darzustellen und die notwendigen Übergangsfunktionen von einer Ebene zu einer anderen zu integrieren.

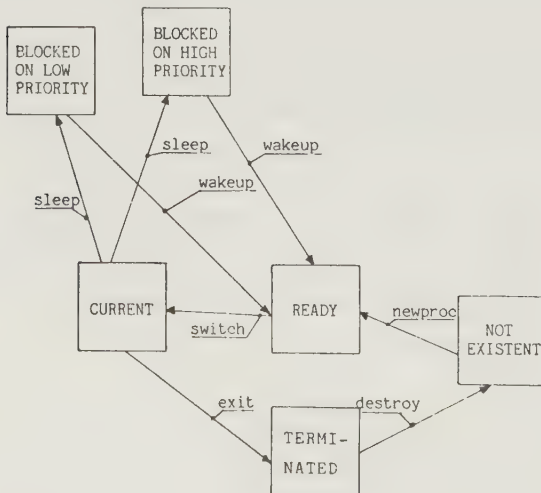


Abb. 5.2.1 Zustandsdiagramm - Monoprozessorsystem

In Abb. 5.2.2 ist der Weg beschrifteten worden, das Zustandsdiagramm für Prozessoren in Polyprozessorsystemen aus der Sicht eines Prozessors darzustellen. Gezeigt werden hier nur diejenigen Prozeßzustände, die direkt von Prozessen, die zum betrachteten Zeitpunkt auf diesem Prozessor residieren, erreicht werden können. Wie leicht einzusehen ist, ist durch geeignete Aneinanderreihung von Übergangsfunktionen jeder Zustand im Gesamtzustandsraum erreichbar.

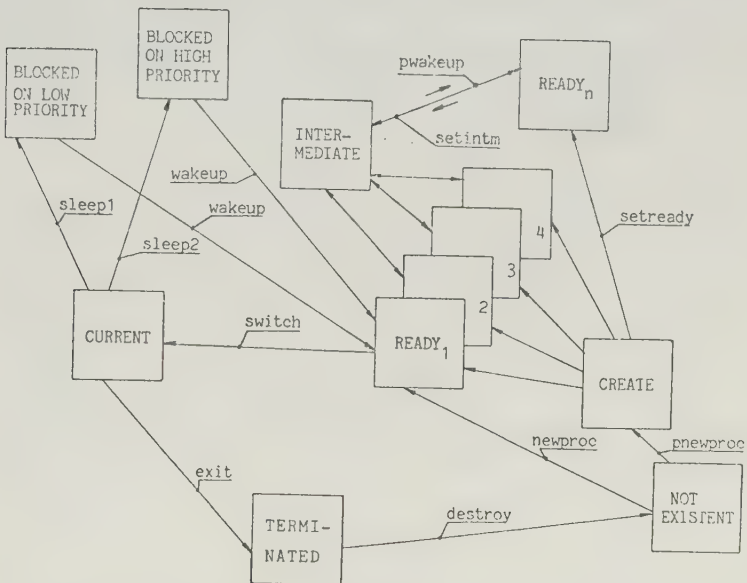


Abb. 5.2.2 Zustandsdiagramm - Polyprozessorsystem

Da es nicht sinnvoll ist, jedem Prozeß, der vom System abgewickelt wird, alle möglichen Zustandswechsel zu gestatten, ist es notwendig, eine Klassifikation der Prozesse einzuführen:

Def. 5.1: Prozesse, die nur auf den Prozessoren lauffähig sind, auf denen sie erzeugt wurden, heißen lokal.

Def. 5.2: Prozesse, deren Umschalten auf einen anderen Prozessor den Transfer mindestens eines Segments nach sich zieht, heißen global/temporär lokal.

Def. 5.3: Prozesse, deren Umschalten auf einen anderen Prozessor ohne Kopiervorgänge abzuwickeln ist, heißen global.

Im folgenden werden die Prozeßklassen mit 'procl', 'procgt' bzw. 'procg' abgekürzt.

Die Fähigkeit eines Prozesses, seine Residenz im System von einem Prozessor auf einen anderen zu verlagern, hängt im wesentlichen von der Plazierung und der Organisation seiner einzelnen Segmente ab. Daraus folgt, daß für die einzelnen Prozeßklassen nur bestimmte Segmenttypkombinationen zulässig sind (Kap. 5.3).

5.2.1.2 Erzeugen eines Prozesses

Das Erzeugen eines Prozesses (Siehe auch: Kap. 2.2.2) wird mit Hilfe des Systemdienstes 'fork' für die lokalen Prozesse und 'pfork' (Kap. 6.1) für die globalen Prozesse durchgeführt. Der neu in das System eingeschleuste Prozeß (Sohn) ist eine Kopie des generierenden Prozesses (Vater). Der neue Prozeß kann nun das auf einem File residierende Ladeobjekt mit Hilfe des Betriebssystemdienstes 'exec' laden und

ausführen. Hierbei werden die ursprünglichen Segmente mit den neuen überladen. Ab diesem Zeitpunkt kann der Sohnprozeß nicht mehr auf seine bisherigen Segmente zugreifen, es sei denn, es handelt sich um gemeinsam benutzte Segmente.

Für lokale Prozesse laufen die Vorgänge beim Erzeugen eines Prozesses und beim Ausführen eines Programms unverändert wie in einem Monoprozessorsystem ab. Beim Generieren von Prozessen der Klasse 'procg' und 'proctg' besteht die Möglichkeit, die Prozeß/Prozessorzuordnung zu steuern. Im Betriebssystemdienst 'pfork' wird in einem Parameter die Prozessoraffinität des Prozesses angegeben. Die Prozessoraffinität gibt die Menge der Prozessoren an, auf denen der Prozeß ablauffähig sein soll. Ist der Parameter leer, so umfaßt die Prozessoraffinität die Menge aller Prozessoren des Systems. Sollen gemeinsame Segmente von Prozessen auf verschiedenen Prozessoren betrieben werden, so müssen in jedem Fall diese Prozesse mit dem Betriebssystemdienst 'pfork' erzeugt werden, auch dann, wenn die einzelnen Prozesse jeweils nur auf einem bestimmten Prozessor residieren sollen.

5.2.1.3 Plazierung der Segmente

Bei der Organisation der Prozeßverteilung über das gesamte System, spielt die Plazierung der Segmente eine wesentliche Rolle, so daß hier im Vorgriff auf Kap. 5.3 einiges über die Strategie bei der Plazierung der Segmente gesagt werden soll.

Die Segmentverwaltung trachtet bei der Plazierung der Segmente danach, einen hohen Grad an Konfliktfreiheit am gemeinsamen Speicherbus zu erreichen. Soweit irgend möglich werden also die Segmente in die lokalen Speicher plaziert. Dies kann so weit gehen, daß gemeinsame Codesegmente in mehreren Kopien in den einzelnen Speichern vorhanden sind.

Will man jedoch extrem schnelle Umschaltzeiten beim Wechsel eines Prozesses von einem Prozessor auf einen anderen erreichen, so kann man den Binder (Kap. 6.2) veranlassen, die Segmente geeignet zu kennzeichnen. Der Benutzer muß sich jedoch darüber im Klaren sein, daß bei intensiver Anwendung der Prozeßklasse 'procg' die Verarbeitungsgeschwindigkeit trotz hohem Parallelitätsgrad durch die Flaschenhalswirkung des gemeinsamen Speichers nicht mehr gesteigert werden kann. Durch die gegenseitige Behinderung kann es sogar zu einer Leistungsverminderung ab einer bestimmten Anzahl von Prozessoren kommen.

Der 'Overhead', der bei einem Prozessorwechsel der Prozeßklasse 'procgt' entsteht, wird von der Kopierzeit der lokalen Segmente bestimmt. Ein Prozessorwechsel eines blockierten und ausgelagerten Prozesses verursacht keinen zusätzlichen 'Overhead', wenn man den Hintergrundspeicher analog dem Arbeitsspeicher organisiert. Die Voraussetzungen für die Organisation eines solchen Hintergrundspeicherkonzepts sind durch das Kommunikationssystem und durch die Bussystemverwaltung (Kap. 5.6.3) gegeben. Die Implementierung dieses Moduls soll jedoch nicht Gegenstand dieser Arbeit sein, so daß hier nur der im folgenden Abschnitt beschriebene 'innere Prozeßwechsel' mit dem notwendigen Kopieren der lokalen Segmente betrachtet wird.

5.2.1.4 Prozeßwechsel/Prozessorwechsel

Der Prozeßwechsel aus lokaler Sicht hat sich gegenüber dem Monoprozessorsystem nur in soweit geändert, daß für die einzelnen Prozeßklassen verschiedene Prozeßkontrollblöcke notwendig sind und daß die Zugriffe auf die Prozeßkontrollblöcke der Prozeßklassen 'procg' und 'procgt' unter gegenseitigem Ausschluß erfolgen.

Der Übergang eines Prozesses auf einen anderen Prozessor wird vom System dann angestoßen, wenn es Prozessoren gibt, die lokal über keine lauffähigen Prozesse mehr verfügen. Prozessoren mit dieser Eigenschaft (Anforderungsprozessoren) suchen sich nun den Prozeß aus der Menge der auf ihm lauffähigen aber nicht residierenden Prozesse (Auswahlmenge) mit der höchsten Priorität aus und senden dem Prozessor auf dem der gefundene Prozeß residiert (Residenzprozessor) eine Nachricht, die ihn veranlasst, den gewünschten Prozeß in den 'intermediate'-Status zu überführen. Der Residenzprozessor veranlaßt diesen Übergang unter der Voraussetzung, daß dadurch nicht eine vorgebbare Anzahl lauffähiger Prozesse der Klassen 'procg' oder 'procgt' unterschritten wird. (Ohne diesen Schwellwert besteht die Gefahr, daß sich die Prozessoren bei ungünstiger Systemauslastung ständig gegenseitig die Prozesse wegnehmen.) Eine Quittung an den Requestprozessor zeigt diesem an, ob er den gewünschten Prozeß übernehmen kann oder nicht. Im letzteren Fall wird der oben beschriebene Vorgang wiederholt, wobei der Prozeß des vorhergehenden Versuchs nicht mehr Element der Auswahlmenge ist. Ist die Auswahlmenge leer, so geht der Prozessor in den Zustand 'idle' über.

Der Modul, der die Funktionen für einen Prozessorwechsel eines Prozesses zur Verfügung stellt, ist derart gestaltet, daß der interne Strategiemodul leicht austauschbar ist, so daß später - falls erforderlich - leicht durch aufwendigere Strategien die Systembalance verbessert werden kann. Da die Systembalance stark vom Aufgabenprofil der Benutzersysteme abhängen wird, ist es denkbar, den Strategiemodul zu einer Strategiebibliothek zu erweitern. Hierdurch wäre es dem Betriebssystem möglich, je nach Lastverhalten, dynamisch die Prozessorvergabestrategie zu wechseln.

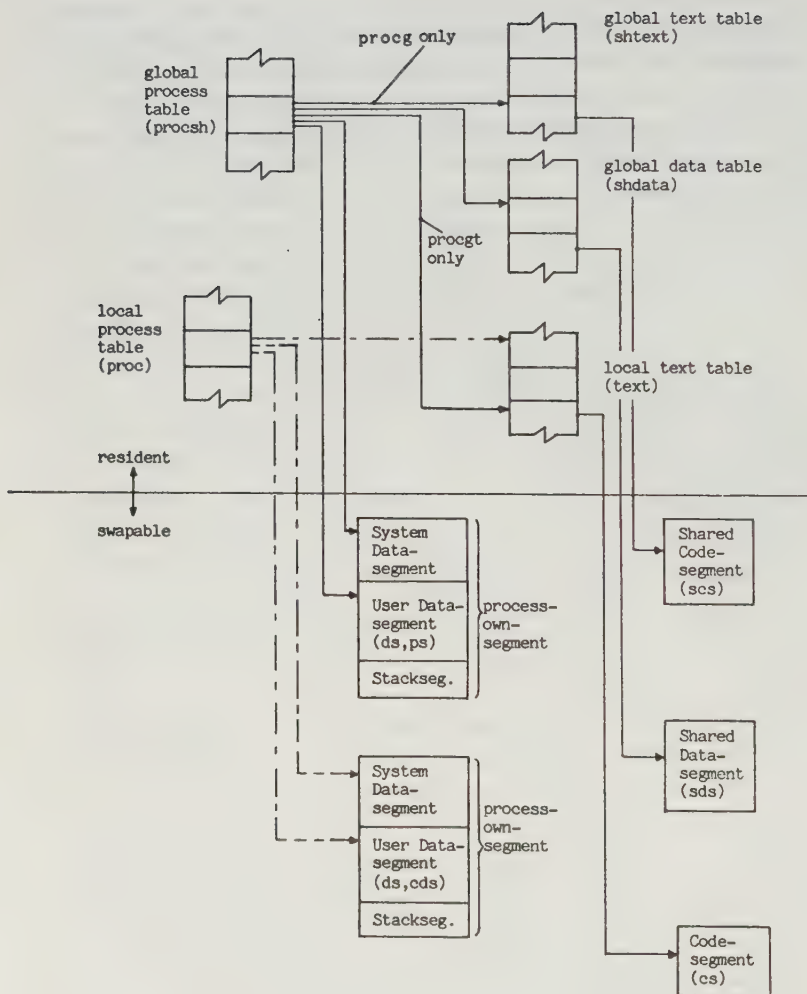


Abb. 5.2.3 Prozeßkontrolldatenstrukturen

5.2.1.5 Prozeßkontrolldatenstrukturen

In Abb. 5.2.3 wird ein Überblick über die Kontrolldatenstrukturen der Prozeßverwaltung und der Segmentverwaltung gegeben. Auch aus dieser Abbildung sind die engen Wechselbeziehungen zwischen beiden Moduln erkennbar, da, wie schon erwähnt, die Art und Platzierung der Segmente die Prozeßklasse bestimmen. Die durchgezogenen Linien zeigen die Verkettung der Kontrolldatenstrukturen für die Prozeßklassen 'prog' und 'progt', während die gestrichelten Linien die Struktur eines lokalen Prozesses wiedergibt. Diese Struktur ist identisch mit der des Monoprozessorsystems.

Der auslagerbare Teil des Prozeßkontrollblocks, das 'System Data Segment', das Datensegment und das Stacksegment werden in einem Arbeitsspeicherblock abgelegt ('Process Own Segment'). Gehören diese Segmente einem Prozeß der Klasse 'prog', so werden sie in den gemeinsamen Speicher platziert.

5.2.2 Spezifikation

```
MODULE prozessverwaltung /procm/
```

TYPES

```
puids:      {FOR i FROM 0 TO (maxpu-1): 2i };
                * Prozessoridentifikation
```

```

pumasks:      {FOR j FROM 0 TO (maxpu-1):  $\sum_{i=0}^{i=\text{maxpu}-1} 2^i - 2^j$  };

```

* Prozessormasken

```

puaff:
    {  $\sum_{i=1}^{|P_j|} p_i \mid p_i \in P_j \wedge p_j \in \text{POTENZ}(\text{puids})$  };

    * Prozessoraffinität

```

```

procstat:    {cu,          * current
              re,          * ready
              bhp,         * blockinhighprio
              blp,         * blockinlowprio
              intm,        * intermediate
              term,        * terminated
              crea         * create
              };          * Prozessstatus

```

```
proctype: {procl,      * lokaler Prozess
          proegt,     * global/temp.lokaler Prozess
          procg       * globaler Prozess
        };           * Prozessklassen
```


PARAMETERS

INTEGER maxpu, * maximale Anzahl der Prozessoren
 nproc, * maximale Anzahl der Prozesse der
 Klasse 'procl'
 nprocsh; * maximale Anzahl der Summe der Prozesse
 der Klassen 'procg' und 'procgt'

Bemerkung: Im folgenden werden im Kommentar die Begriffe
Prozess und Prozesskontrollblock synonym verwendet.

 ready: {pcb | pcb.stat = re};
 * Menge der laufbereiten Prozesse

 ready_{puid}: {pcb ∈ ready ∧ pcb.cproc = puid};
 * Menge der laufbereiten Prozesse bezüglich
 des Prozessors 'puid'

 intermediate: {pcb | pcb.stat = intm}
 * Menge der Prozesse im Zustand 'inter-
 mediate'

 intermediate_{puid}: {pcb ∈ intermediate ∧ pcb.cproc = puid};
 * Menge der Prozesse im Zustand 'inter-
 mediate' bezüglich des Anforderungs-
 prozessors 'puid'

 current: {pcb | pcb.stat = cu};
 * Menge der laufenden Prozesse

 pcb_c: {pcb | pcb ∈ current ∧ pcb.cproc = c};
 * der auf dem betrachteten Prozessor lau-
 fende Prozess

 pool_{puid}: {pcb ∈ ready ∧ pcb.cproc ≠ puid ∧
 puid ∈ pcb.puaffin};
 * Menge der laufbereiten, aber nicht auf
 Prozessor 'puid' residierenden Prozesse

```
pool: UNION pooli , 0 ≤ i ≤ maxpu;  
      * Vereinigungsmenge der Auswahlmengen  
blowprio: {pcb | pcb.stat = blp};  
      * Menge der blockierten Prozesse auf  
        niedriger Prioritätsstufe  
bhighprio: {pcb | pcb.stat = bhp};  
      * Menge der blockierten Prozesse auf  
        hoher Prioritätsstufe  
proc: {pcb | pcb.type = procl};  
      * Menge der Prozesse der Klasse 'procl'  
procsh: {pcb | pcb.type = procg ∨  
            pcb.type = proclt}  
      * Die Vereinigungsmenge der Prozesse der  
        Klassen 'proclt' und 'procg'  
idle: {pu | pu.stat = idle};  
      * Menge der Prozessoren ohne zugeteilten  
        Prozess
```

DEFINITIONS

```
pcbb(chan) IS {pcb | pcb ∈ (blowprio UNION bhighprio)  
                ∧ pcb.chan = chan};  
      * liefert die Menge der block. Prozesse,  
        deren Deblockierereignis mit 'chan'  
        codiert ist  
pcbb1 IS {pcb | pcb ∈ (blowprio UNION bhighprio)  
          * liefert die Menge aller block. Prozesse
```

```

pcb max(xxxxx,puid) IS {pcbj | pcbj ∈ xxxxxpuid ∧
    (¬ ∃ pcbi | pcbi ∈ xxxxxpuid ∧
        pcbi.prio > pcbj.prio),
    1 ≤ i,j ≤ |xxxxxpuid|};
    * maximales Element aus der Menge
      xxxxxpuid gemäss einer Prioritäts-
      relation
pcb min(pool) IS {poolj | (∃ pooli |
    |pooli| > |poolj|), 1 ≤ i,j ≤ maxpu};
    * minimales Element aus der Vereini-
      gungsmenge pool gemäss der Mächtigkeit
      der Mengen pooli
puids puid IS {pucbtabs(puc).me};
    * Prozessoridentifikation des 'current'
      Prozessors

```

FUNCTIONS

```

OFUN sleep(chan,pri);

```

```

    * Parameter: chan - Ereigniscode; alle Prozesse, die
      auf das gleiche Ereignis warten,
      werden 'am' gleichen Code blok-
      kiert
      pri - Priorität des zu blockierenden
      Prozesses
    * Effekt : Blockiert den aufrufenden Prozess

```

EXCEPTIONS

```

    none;

```

EFFECTS

```

    IF pri ≥ 0
    THEN pcbc ∈ blowprio;
    ELSE pcbc ∈ bhighprio;

```

OFUN switch();

* Effekt: switch teilt einem gemäss Auswahlpolitik
ausgewählten Prozess den Prozessor zu

EXCEPTIONS

none;

EFFECTS

TYPECASE ready_{puid} OF

≠ ∅: pcb_c = pcb | max(ready,puid);

= ∅: IF pool_{puid} ≠ ∅
THEN

WHILE (pool_{puid} ≠ ∅ ∧
setintm(puid,max(pool,puid)) = false)
pool_{puid} \ max(pool,puid);

pwakeup;

IF ready_{puid} ≠ ∅

THEN pcb_c = pcb | max(ready,puid);

ELSE puid ∈ idle;

ELSE puid ∈ idle;

OFUN wakeup(chan);

* Parameter: chan - Ereigniscode; alle Prozesse, die
auf das gleiche Ereignis warten,
werden 'am' gleichen Code block-
tiert

* Effekt : Alle Prozesse, die an 'chan' blockiert
sind, werden deblockiert

EXCEPTIONS

none;

EFFECTS

∀ pcb ∈ (pcb_b(chan) INTER proc) : setrun(pcb);

∀ pcb ∈ (pcb_b(chan) INTER procsh): psetrun(pcb);

```
VFUN testchan(chan) → test;
    * Parameter: chan - Ereigniscode
                        test = TRUE, wenn ein Prozess existiert,
                        der an 'chan' blockiert ist
                        test = FALSE; sonst
```

EXCEPTIONS

none;

INITIALLY FALSE;

OFUN setrun(pcb);

* Parameter: pcb - zu deblockierender Prozess

* Effekt: Überführt den blockierten Prozess pcb @
proc in den Zustand 'ready'

HIDDEN;

EXCEPTIONS

pcb ↗@ { pcb_{bl} INTER proc };

EFFECTS

pcb @ ready;

OFUN psetrun(pcb);

* Parameter: pcb - zu deblockierender Prozess

* Effekt: Überführt den blockierten Prozess
pcb @ procsh in den Zustand 'ready'

HIDDEN;

EXCEPTIONS

pcb ↗@ { pcb_{bl} INTER procsh };

EFFECTS

pcb @ ready;

OVFUN setintm(apuid,pcb) → ok;

* Parameter: apuid - Anforderungsprozessor
 pcb - angeforderter Prozess
 ok=TRUE - angeforderter Prozess befindet sich im Zustand 'intermediate'
 ok=FALSE - sonst
 * Effekt: Setzt den angeforderten Prozess in den Zustand 'intermediate'

EXCEPTIONS

pcb ∈ {{ready_{puid} INTER procsh} |
 {{ready_{puid} INTER procsh}} > limitsh};

EFFECTS

pcb ∈ intermediate_{apuid};
 ok = TRUE;

OFUN pwakeup();

* Effekt: Überführt Prozesse aus dem Zustand 'intermediate' in den Zustand 'ready'

EXCEPTIONS

{ intermediate INTER procsh INTER pool_{puid} } = ∅;

EFFECTS

∀ pcb' ∈ intermediate_{puid}: pcb ∈ ready_{puid};

OVFUN pnewproc(procaff,procclass) → pid;

- * Parameter: procaff - Prozessoraffinität
- proctype - Prozessklasse
- pid - systemeindeutige Prozess-
 identifikation
- * Effekt: Generiert eine neue Prozessumgebung

EXCEPTIONS

procclass ∈ proc;

|procsh| ≥ nprocsh;

EFFECTS

|procsh| = |procsh'| + 1;

|create| = |create'| + 1;

pid = new(pid);

OVFUN newproc() → pid;

- * Parameter: pid - systemeindeutige Prozessidenti-
 fikation
- * Effekt: Generiert eine neue Prozessumgebung

EXCEPTIONS

proctype ∈ procsh;

|proc| ≥ nproc;

EFFECTS

|proc| = |proc'| + 1;

|ready| = |ready'| + 1;

pid = new(pid);

OFUN setready(pcb);

* Effekt: Überführt den Prozess pcb vom Zustand
'create' in den Zustand 'ready'

EXCEPTIONS

pcb $\rightarrow$ pool_{puid};

EFFECTS

pcb $\in$ ready_{puid};

VFUN currenttype $\rightarrow$ procclass;

* Parameter: - procclass enthält die Prozessklasse
des Prozesses im Zustand 'current'

EXCEPTIONS

none;

INITIALLY UNDEFINED;

VFUN currentfathertype $\rightarrow$ procclass;

* Parameter: - procclass enthält die Prozessklasse
des Vater(erzeugenden)-Prozesses

EXCEPTIONS

none;

INITIALLY UNDEFINED;

VFUN currentpcb $\rightarrow$ j;

* Parameter: - j enthält den Index des Prozess-
kontrollblocks des Prozesses im Zustand
'current'

EXCEPTIONS

none;

INITIALLY UNDEFINED;

VFUN currentproc → procid;

* Parameter: - procid enthält die Prozessoridentifikation des 'current' Prozessors

EXCEPTIONS

none;

INITIALLY UNDEFINED;

VFUN gcpce → i;

* Parameter: - i enthält den Index der lokalen Zugriffsliste des Prozesses im Zustand 'current'

EXCEPTIONS

none;

INITIALLY UNDEFINED;

Folgende Funktionen bleiben unverändert:

OFUN exit;

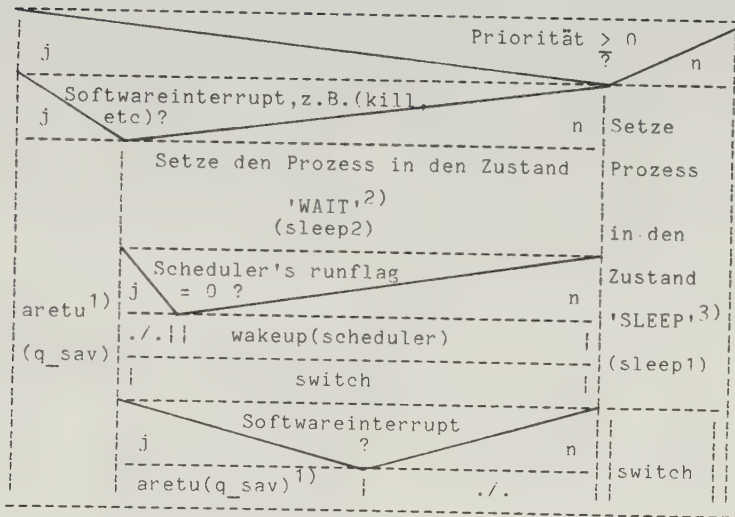
OFUN destroy;

END_MODULE

5.2.3 Implementierung

sleep(chan):

Parameter: chan - Weckereigniscode des zu blockierenden Prozesses

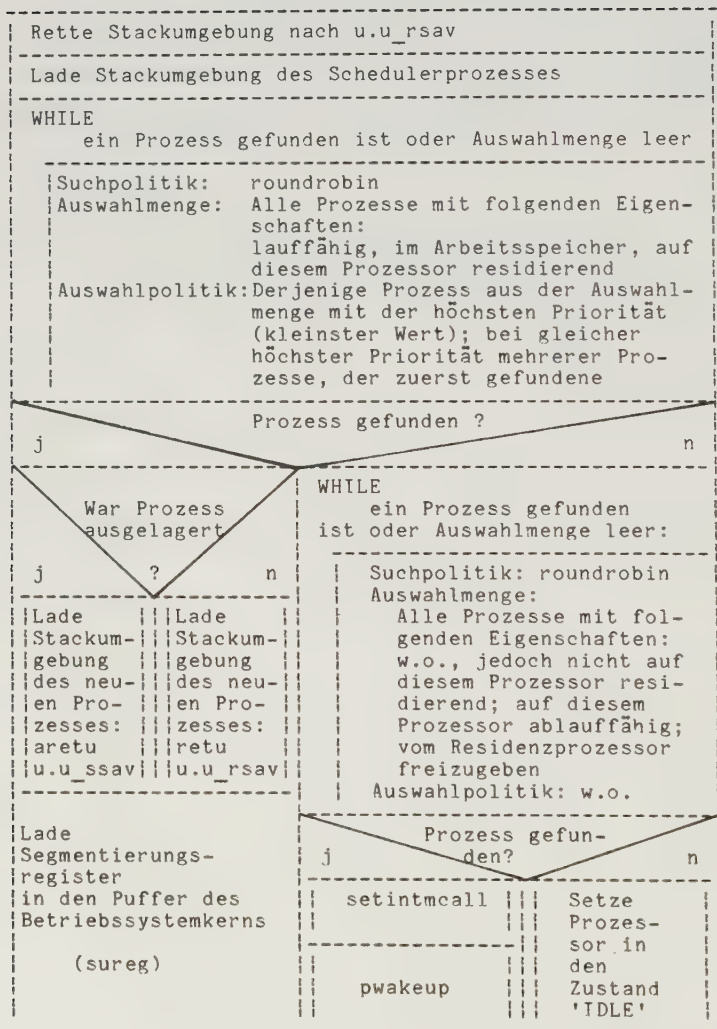


1) Lade Stackumgebung (Stackzeiger und Stackbasis) aus 'q_sav'

2) 'WAIT' $\hat{=}$ BLOCKED ON HIGH PRIORITY

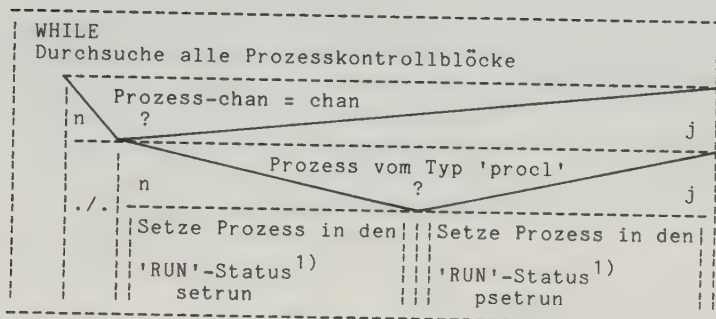
3) 'SLEEP' $\hat{=}$ BLOCKED ON LOW PRIORITY

switch:



wakeup(chan):

Parameter: chan - Weckereigniscode des blockierten Prozesses



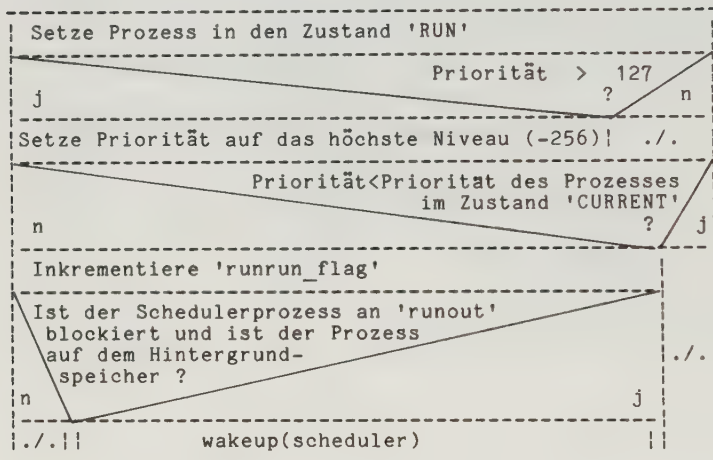
1) READY ist die Teilmenge der Prozesse im Zustand 'RUN', die sich im Arbeitsspeicher befindet.

setrun(p)/psetrun(p):

setrun: Prozessklasse 'procl'

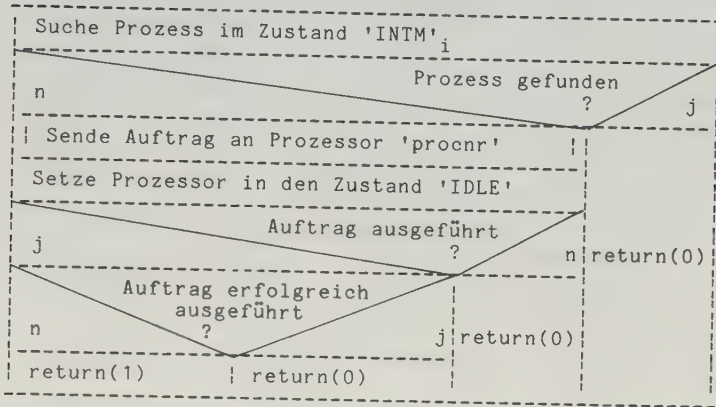
psetrun: Prozessklasse 'procg' und 'procg'

Parameter: p - Zeiger auf Prozesskontrollblock



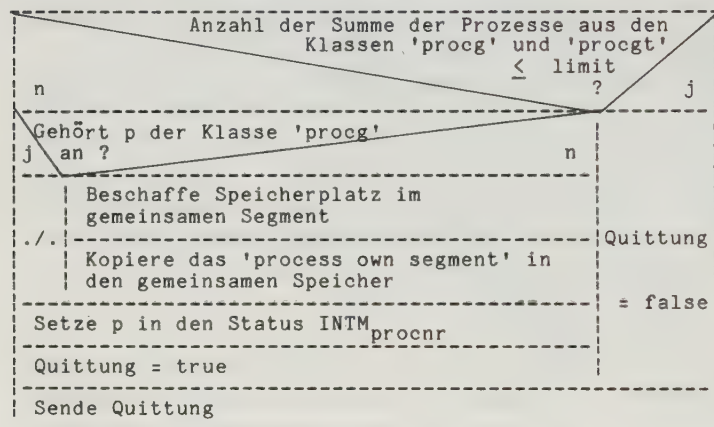
setintmcall(procnr,p)

Parameter: procnr - Prozessornummer des Residenzprozessors
 p - Zeiger auf Prozesskontrollblock

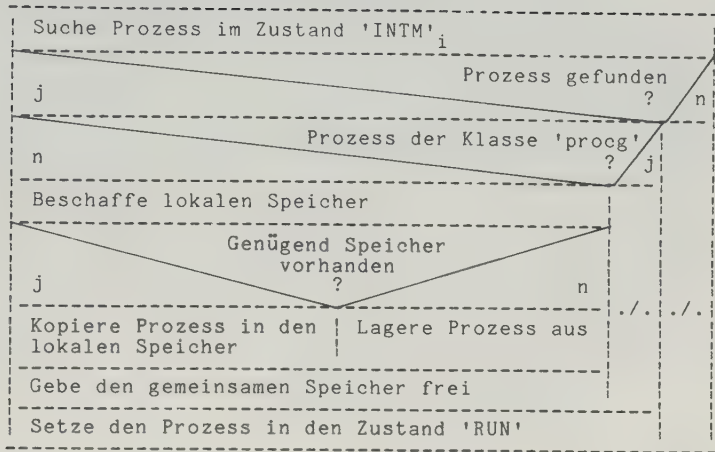


setintm(procnr,p)

Parameter: procnr - Prozessornummer des Anforderungs-
 prozessors
 p - Zeiger auf Prozessorkontrollblock

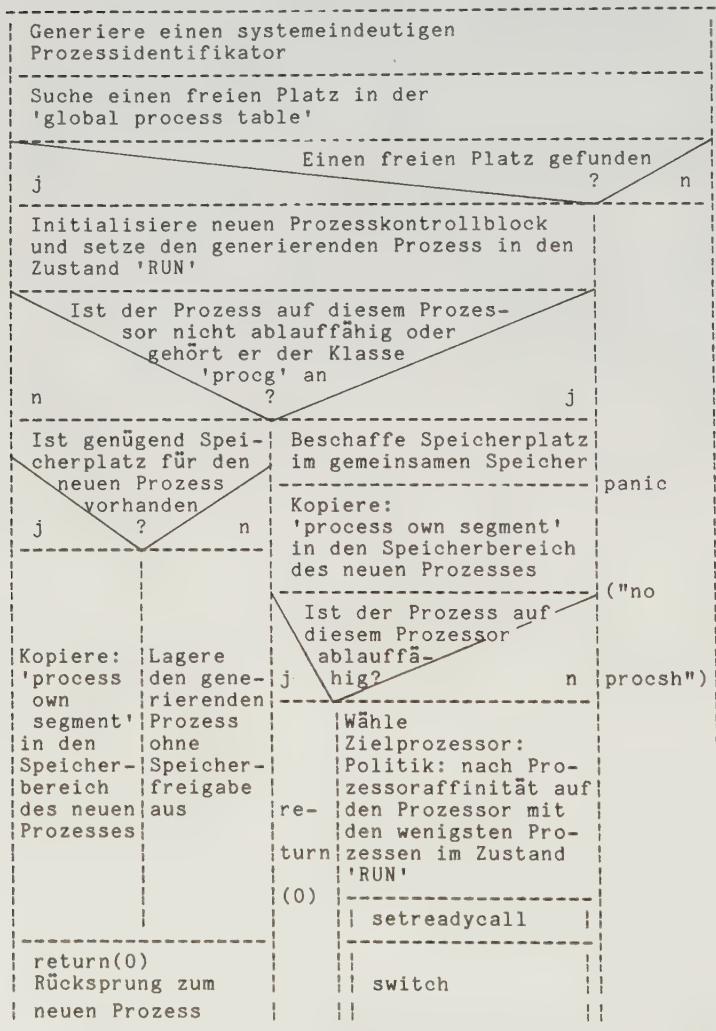


pwakeup:



pnewproc(procaff,proctype)

Parameter: procaff - Angabe, auf welchen Prozessoren der neue Prozess ablauffähig ist
 proctype =0 Prozess gehört der Klasse 'procgt' an
 =1 Prozess gehört der Klasse 'procg' an

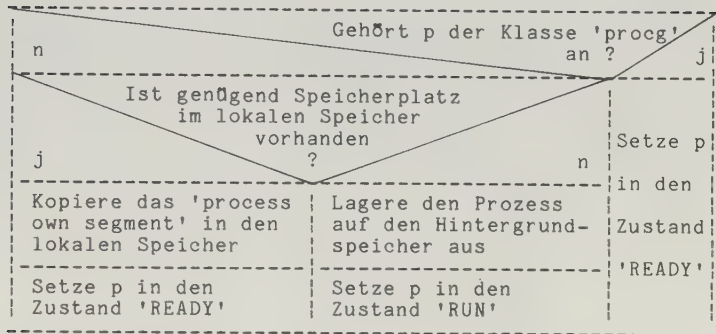


newproc()

bleibt unverändert

setready(p)

Parameter: p - Zeiger auf Prozesskontrollblock



5.3 Segmentmanagement

5.3.1 Überblick

Das Segmentmanagementsystem umfaßt folgende Betriebssystemmoduln:

- Segmentverwaltung

Hier sind die Funktionen zur Verwaltung der Segmentkontrollblöcke implementiert. Die Segmentverwaltung ist im wesentlichen für die segmenttypspezifische 'Buchführung' über Zustand, Ort, Typ, Referenzen, Bindungen an Prozesse, etc. zuständig. Außerdem werden von hier unter bestimmten Voraussetzungen Funktionen des 'Swappers' angestoßen.

- Objektschutzsystem

Das Objektschutzsystem ist für die Vergabe von Zugriffsrechten auf Segmente und für deren Einhaltung verantwortlich.

- Scheduler

Der 'Scheduler' organisiert nach vorgegebener Strategie den Segmenttransfer innerhalb des Speichersystems.

- Swapper

Der 'Swapper' führt den physikalischen Transfer der Segmente zwischen Arbeitsspeicher und Hintergrundspeicher durch. Er realisiert die Schnittstelle zwischen dem Segmentmanagement und dem E/A-System.

- Lader

Der Lader schleust die Segmente in das Segmentmanagement ein.

Für die Verwaltung der Segmente wird eine Klassifikation gemäß Tab. 5.3.1 eingeführt.

<u>Typ</u>	<u>Eigenschaften</u>	<u>Schutz- schlüssel</u>
Code/Datensegment (cds)	nicht reentrant, lokaler Speicher	ex, w, r
Codesegment (cs)	reentrant, lokaler Speicher	ex
Datensegment (ds)	nur Daten, lokaler Speicher	w, r
'Shared'-Codesegment (scs)	reentrant, gemeinsamer Speicher	ex
'Shared'-Datensegment (sds)	nur Daten, gemeinsamer Speicher	sdsr: r, sdsd: w, r
'Privates-Datensegment' (ps)	nur Daten, gemeinsamer Speicher	w, r
Stack-Segment(sts)	nur Daten Speicher abhängig von Prozeßklasse	w, r

Tab. 5.3.1 Klassifikation der Segmente

Aus den in Tab. 5.3.2 aufgeführten zulässigen Kombinationen der einzelnen Segmenttypen, leiten sich die verschiedenen Prozeßklassen ab, die ebenfalls bei der Prozeßverwaltung des Systems von Bedeutung sind. Die Platzierung des Stacksegments

wird in Abhängigkeit von Datensegment vorgenommen. Es wird in den Fällen 1 - 5 im lokalen Speicher, im Fall 6 im gemeinsamen Speicher angelegt. Die Zusammenfassung der Segmenttypen 'eds' und 'sts', 'ds' und 'sts' bzw. 'ps' und 'sts' mit dem zum Prozeß gehörenden 'System Data Segment' zu einem Arbeitsspeicherblock wird als 'Process Own Segment' bezeichnet. (Kap. 5.2)

<u>Segmentkombination</u>	<u>Prozeßklasse</u>	<u>Abkürzung</u>
1. eds	lokal	procl
2. cs + ds	lokal	procl
3. cs + ds + sds	lokal	procl
4. cs + ds + sds	global/temp.lokal	procgt
5. cs + sds	global/temp.lokal	procgt
6. scs + sds + ps	global	procg

Tab. 5.3.2 Klassifikation der Prozesse

5.3.2 Segmentverwaltung

Anhand des Segmentzustandsdiagramms sollen nun die einzelnen Grundfunktionen des Moduls erläutert werden.

5.3.2.1 alloc - Binden eines Prozesses an ein Segment

Die Funktion alloc1 wird während der Ladephase eines Segments ausgeführt. Sollte bei den Segmenttypen 'cs', 'scs' oder 'sds' dieses Segment schon im Hintergrundspeicher oder Arbeitsspeicher vorhanden sein, so wird die Funktion alloc2 bzw. alloc3 ausgeführt.

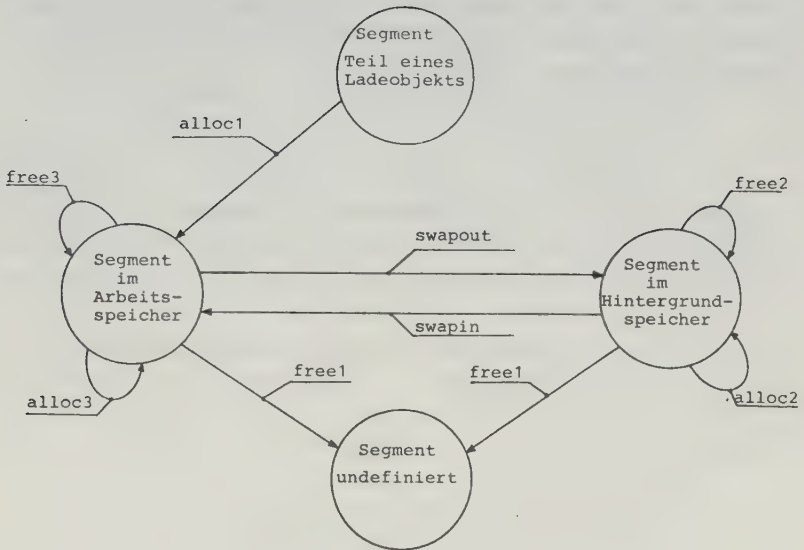


Abb. 5.3.1 Segmentzustandsdiagramm

5.3.2.2 free - Auflösung der Bindung: Prozeß - Segment

Die Funktionen free3 bzw. free2 lösen die Bindung Prozeß - Segment je nach Lage des Segments im Arbeitsspeicher oder im Hintergrundspeicher. Für die Segmente vom Typ 'cds', 'ds' und 'ps' ist die Ausführung der Funktion free1 (Zerstören des Segments) obligatorisch. Für Segmente des Typs 'scs' und 'cs' wird free1 durchgeführt, wenn keine Referenzen mehr auf diese Segmente existieren. Eine besondere Stellung neh-

men die Segmente des Typs 'sds' ein, da hier die Funktion free1 erst dann ausgeführt werden darf, wenn auch in Zukunft (noch zu ladende Prozesse eines parallelen Prozeßsystems) keine Referenz mehr zu erwarten ist.

5.3.3 Adressierungsschema und Objektschutz

In diesem Abschnitt soll dargestellt werden, wie das in diesem System implementierte Schutzsystem in die aus der Literatur bekannten Modelle und Lösungsvorschläge einzuordnen ist.

Dem hier realisierten Objektschutzsystem liegt folgende Definition zugrunde: /Jäpe77/

Definition:

Ein Objektschutzsystem besteht aus

- a.) einer Menge O von Objekten,
- b.) einer Menge S von Subjekten,
- c.) einer Menge Z von Zugriffsrechten, welche die Subjekte auf Objekte ausüben können,
- d.) einem Regelsystem, das festlegt, welches Subjekt in welcher Weise auf welches Objekt zugreifen darf.

Zu diesem oben definierten Objektschutzsystem gehört ein Mechanismus, der die Einhaltung der Regeln des Regelsystems garantiert. Diese Instanz soll als Schutzmechanismus bezeichnet werden.

Der Schutzmechanismus identifiziert jede beabsichtigte Handlung eines Subjekts und gestattet die Ausführung der Handlung nur dann, wenn nach dem geltenden Regelsystem die Handlung ausgeführt werden darf. Die Zugriffsrechte, die Subjekte auf Objekte besitzen, lassen sich im Regelsystem

als Tripel (Subjektbezeichnung, Zugriffsrecht, Objektbezeichnung) darstellen. Da das Regelsystem nur die erlaubten Handlungen explizit aufzählt, nennt man ein derartiges System auch ein positives, explizites Rechtssystem - es ist grundsätzlich alles verboten, was nicht ausdrücklich erlaubt ist.

Der Wirkungsbereich eines Subjekts ist durch die Menge der Tripel (S, Z, O) im Regelsystem beschrieben, die die Komponente S gemeinsam haben. Dieser Wirkungsbereich wird oft auch als "Schutzumgebung" oder "Domäne" bezeichnet.

Im Kern des Schutzsystems sind diejenigen Subjekte zusammengefaßt, die Zugriffsrechte auf das Objekt "Regelsystem" besitzen. Hier werden die Entscheidungen getroffen, ob und in welcher Form, Subjekten Zugriffe auf Objekte gestattet werden können.

Bei der Realisierung eines Schutzsystems sind die Mechanismen, die von der Hardware eines Rechners angeboten werden, von entscheidender Bedeutung. Als Voraussetzung für eine effiziente Implementierung muß der Schutzmechanismus in der Hardware realisiert sein. Desweiteren sollten schnelle speichernde Hardwareelemente vorhanden sein, in denen mindestens die aktuellsten Teile eines Regelsystems abgelegt werden können.

In der Hardware der Prozessoren des Perkin Elmer 7/32-Systems sind Voraussetzungen für die Realisierung eines virtuellen, segmentierten Adreßraums geschaffen worden. Pro Prozessor existiert eine Segmenttabelle mit 16 Tripeln von Registern. Ein Tripel besteht aus einem Basisregister, einem Grenzregister und einem Zugriffsschlüsselregister (B, G, Z) .

Das Basisregister führt die einstufige Adreßumsetzung der virtuellen in die physikalische Adresse aus. Der virtuelle und der physikalische Adreßraum umfaßt ein Megabyte. Das Grenzregister stellt sicher, daß der erlaubte Zugriff nur innerhalb des referierten Objekts ausgeführt wird, während

das Zugriffsschlüsselregister die Art des erlaubten Zugriffs kennzeichnet.

Vier verschiedene Zugriffsarten sind realisiert:

- nur ausführen erlaubt
- nur lesen erlaubt
- schreiben und lesen erlaubt
- kein Zugriff erlaubt

Bei der Realisierung eines Schutzsystems kann man nun für das oben beschriebene Regelsystem zwei Lösungsmöglichkeiten erwägen. Eine Möglichkeit wäre eine Gruppierung nach der Objektkomponente durchzuführen. Diese Lösung wurde im MULTICS-Betriebssystem /Salt74/ bei der Realisierung der 'access-lists' beschritten. Der hier eingeschlagene Weg folgt einem Vorschlag von Jones /Jones73/, der in ähnlicher Weise schon früher von Dennis und van Horn gemacht wurde, bei dem eine Gruppierung nach der Subjektkomponente durchgeführt wird¹. Die Liste der Paare (O,Z) soll im folgenden als Zugriffsliste bezeichnet werden.

Bedingt durch die 16 Registersätze in der Segmenttabelle, ist die Anzahl der Objekte (Segmente) auf die ein Subjekt (Prozeß) Zugriffsrechte besitzen kann, auf 32 beschränkt, hiervon können maximal 16 Objekte vom Benutzer selbst erzeugt worden sein.

Der Schutzsystemkern realisiert folgenden Schutz:

1.) Betriebssystem/Benutzersystem

Bei Aufruf eines Betriebssystemdienstes findet grundsätzlich ein Schutzumgebungswechsel statt. Parameter werden auf ihre Zulässigkeit und ihren Wertebereich hin überprüft.

1) durch die Hardware bedingt

2.) Prozeß/Prozeß

Gehören zwei Prozesse keinem gemeinsamen Prozeßsystem an, so hat ein fehlerhaftes Verhalten eines Prozesses nur Auswirkungen auf sein 'process own segment'.

3.) Prozeßsystem/Prozeßsystem

Für Prozeßsysteme, die ein gemeinsames Datensegment betreiben, kann ein fehlerhaftes Verhalten auch Auswirkungen auf das gemeinsame Datensegment haben.

Prozeßsysteme verschiedener Benutzer haben disjunkte Schutzumgebungen.

5.3.3.1 Funktionen des Objektschutzsystems

In Abb. 5.3.2 ist dargestellt, welche Funktionen des Objektschutzsystems für die Verwaltung der Zugriffslisten zur Verfügung stehen.

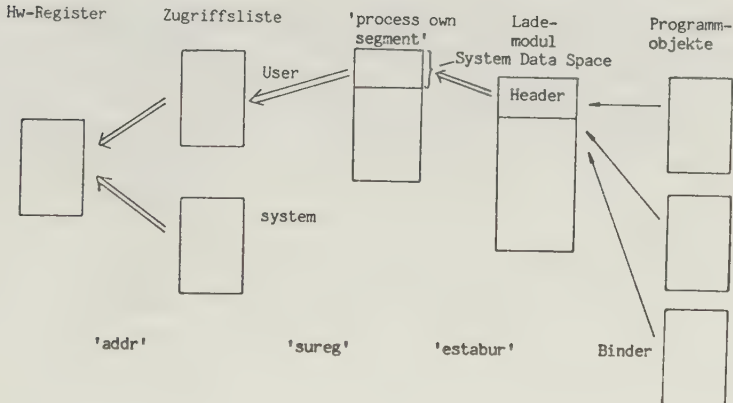


Abb. 5.3.2 Funktionen des Objektschutzsystems

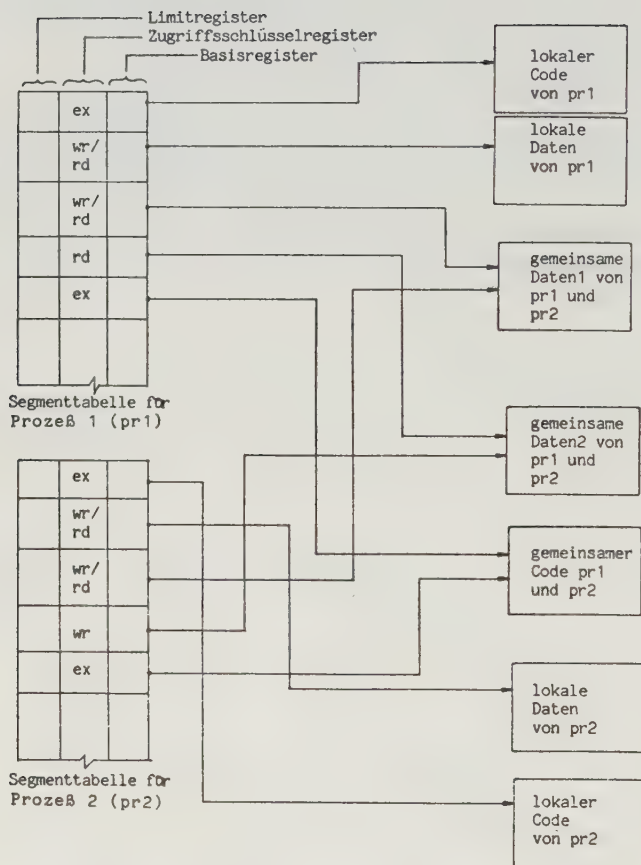


Abb. 5.3.3 Schutzumgebung eines Prozeßsystems

Ein vom Binder generierter Lademodul enthält Angaben über Anzahl, Art und Länge der verschiedenen Segmente. Während der Ladephase wird von der Schutzsystemfunktion 'estabur' aus diesen Angaben ein Prototyp der prozeßspezifischen Zugriffsliste aufgebaut und in das 'System Data Space'-Segment des Prozesses abgelegt. Diese Vorgehensweise hat den Vorteil, daß nur Zugriffslisten geladener Prozesse im Arbeitsspeicher gehalten werden. Der Prototyp der Zugriffsliste enthält alle notwendigen Angaben eines Zugriffslistenelements bis auf die Adresse des Objekts. Wird durch die Prozeßverwaltung einem Prozeß der Prozessor zugeteilt, so ergänzt die Funktion 'sureg' die Prototypzugriffsliste um die Adressen der referierten Objekte und lädt die Liste in den Benutzerzugriffslistenpuffer des Betriebssystemkerns. In einem weiteren Puffer liegt resident die Zugriffsliste des Betriebssystems. Bei einem Schutzumgebungswechsel wird von der Funktion 'addr' die jeweils aktuelle Zugriffsliste in die Segmentierungs- und Schutzregister der Segmenttabelle geladen.

Abb. 5.3.3 zeigt die Schutzumgebung zweier Prozesse eines Prozeßsystems. Für das gemeinsame Datensegment 1 kann das Schutzsystem nur einen Schutz auf Prozeßsystemebene garantieren, da auf dieses Segment beide Prozesse Lese- und Schreibrecht besitzen.

5.3.4 Scheduler - Organisation des Segmenttransfers

Der 'Scheduler', der als lokaler residenter Prozeß implementiert ist, organisiert den Ein/Austransfer der Segmente auf der Basis der "Prozeßumgebung". Es wird also z.B. beim Einlagern von Segmenten nur dann ein Eintransfer angestoßen, wenn sicher ist, daß genügend Platz für sämtliche Segmente, die ein Prozeß referiert, im Arbeitsspeicher vorhanden ist.

Koordiniert wird die Aktivität des Prozesses über die Variablen 'runin' und 'runout'.

runin: Im System sind einlagerbare Prozesse vorhanden; es steht jedoch weder genügend freier Arbeitsspeicher zur Verfügung noch sind auslagerbare Prozesse vorhanden.

runout: Im System sind keine einlagerbaren Prozesse vorhanden.

Angestoßen wird der 'Scheduler' einmal pro Sekunde von der lokalen Uhr, falls er sich an der Variablen 'runin' blockiert hat. Von der Prozeßverwaltung her erfolgen Aufträge z.B. dann, wenn ein Prozeß in den Zustand 'RUN' überführt wurde und sich seine Segmente auf dem Hintergrundspeicher befinden. Koordiniert wird dieser Auftrag über die Variable 'runout'.

Der Schedulerprozeß wählt zunächst aus den auf dem Hintergrundspeicher befindlichen lafbereiten Prozessen denjenigen mit der höchsten Residenzzeit aus. Ist die Menge der lafbereiten und auf dem Hintergrundspeicher befindlichen Prozesse leer, so blockiert sich der Prozeß an der Variablen 'runout'.

Im anderen Fall wird in der Reihenfolge lokaler Arbeitsspeicher - gemeinsamer Arbeitsspeicher versucht, den Anforder-

derungen des ausgewählten Prozesses zu genügen. Mit Hilfe der Funktion 'swapi' werden dann alle benötigten Segmente eingelagert. Scheitert die Beschaffung von genügend lokalem Arbeitsspeicher, so wird aus der Menge der Prozesse, die der Klasse 'procl' angehören, ein auslagerbarer Prozeß bestimmt. Verläuft auch dieser Versuch negativ, so blockiert sich der 'Scheduler' an der Variablen 'runin'.

Bei Speichermangel im gemeinsamen Speicher werden keine Auslagerungsversuche unternommen, da hiervon in der Regel ein ganzes Prozeßsystem betroffen wäre. Hierfür Auslagerungsstrategien zu implementieren, scheint aus Gründen der Komplexität der Auswahlalgorithmen und der notwendigen systemweiten Koordination mit anderen Prozessoren, bei verminderter Wahrscheinlichkeit einen geeigneten Kandidaten zu finden, wenig sinnvoll.

Wird ein auslagerbarer Prozeß gefunden, so wird mit Hilfe der Funktion 'swapout' das 'Process Own Segment' ausgelagert und die Referenzzähler der Segmente des Typs 'cs', 'scs' oder 'sds' dekrementiert. Sind für die zuletzt genannten Segmente keine Ladereferenzen mehr vorhanden, so werden auch diese ausgelagert. Ein echter Datentransfer findet jedoch nur bei Segmenten des Typs 'sds' statt. Der freigewordene Speicher wird an die Freispeicherverwaltung zurückgegeben.

5.3.5 Swapper

In dem Bestreben Strategie und Mechanismus zu trennen, ist für die physikalischen E/A-Transfers der Segmentverwaltung ein eigener Modul, der 'Swapper', implementiert. Da die Schnittstellen des 'Swappers' zum Strategie- und Verwaltungsmodul einerseits und zum E/A-Treibersystem andererseits unverändert geblieben sind, konnte dieser Modul nahezu ohne Modifikationen vom Monoprozessorsystem übernommen werden.

5.3.6 Lader

In der Abb. 5.3.4 ist das Format eines Ladeobjekts mit den für den Lader relevanten Daten dargestellt.

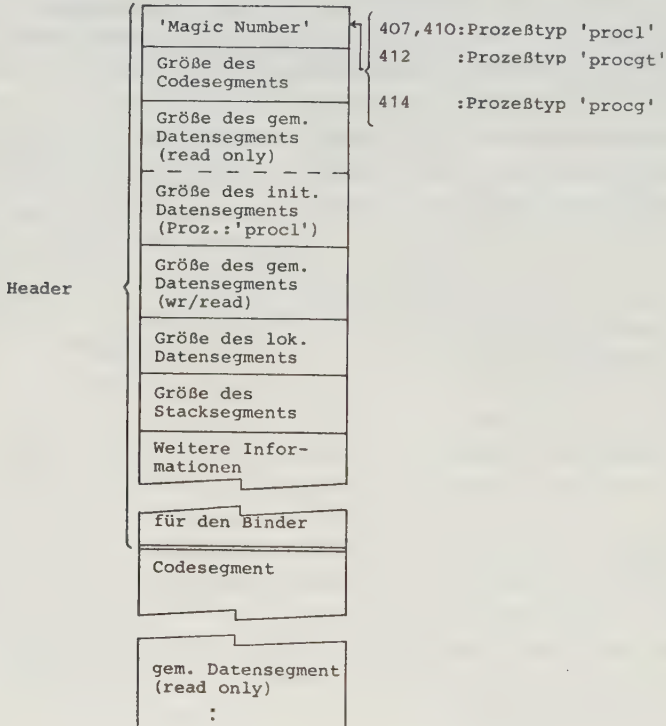


Abb. 5.3.4 Ladeobjektformat

Der Lader ist im Sinne einer Aufwärtskompatibilität so erweitert worden, daß er sowohl Ladeobjekte alter Art, als auch Ladeobjekte neuer Art laden kann. Die sogenannte 'Magic Number' dient dem Lader als Indikator welcher Klasse der ladende Prozeß angehören muß und wie die Kopfinformationen während des Ladens zu interpretieren sind.

5.3.7 Zusammenfassung

Während der 'Swapper' nahezu unverändert vom Monoprozessorbetriebssystem übernommen werden konnte, mußten am 'Scheduler' im wesentlichen nur dort Modifikationen durchgeführt werden, wo prozeßtyp- bzw. segmenttypspezifische Daten referiert werden. Grundlegende Änderungen am Algorithmus des 'Schedulers' waren nicht erforderlich.

Wie nicht anders zu erwarten war, mußte die segmenttypspezifische Verwaltung um wesentliche Teile ergänzt werden.

Die Verwaltung der neu eingeführten Segmenttypen 'sds' bzw. 'scs' wird von den Moduln 'shdata' bzw. 'shtext' wahrgenommen. Bedingt durch die innige Einbettung der gesamten Segmentverwaltung in den Betriebssystemkern war es erforderlich, sich bei der Implementierung stark an den Modul 'text' (Codesegmentverwaltung) anzulehnen, um die Schnittstellen zur Betriebssystemumgebung einhalten zu können. Der neue Modul 'textsh' dient der Verwaltung der lokalen Codesegmente der Prozesse der Klasse 'procgt'; der Algorithmus ist mit dem des Moduls 'text' identisch.

Durch eine umfassende Überarbeitung des Laders konnte der Forderung entsprochen werden, die wesentlich komplexer gewordenen Ladeobjekte neuer Art und die des bisherigen Formats, gleichermaßen bearbeiten zu können.

5.3.8 Spezifikation

5.3.8.1 MODULE Segmentverwaltung / seg /

TYPES

```
sds:      {sds, * Datensegment im gem. Speicher
           ( write-/readable )
           sdsr * Datensegment im gem. Speicher
           ( read only )
           }; * Datenseg. im gem. Speicher

segmenttyp: {cds, * Code/Datensegment
             cs,  * Codesegment
             ds,  * Datensegment
             scs, * Codesegment im gem. Speicher
             sds, * Datensegment im gem. Speicher
             ps,  * Priv. Datensegment im gem. Speicher
             sts, * Stacksegment
             }; * Segmenttypen
```

sgb: STRUCT(

```
segmenttyp type, * Segment
INTEGER loadcount, * Anzahl der an das
                  Segment gebundenen
                  und geladenen Pro-
                  zesse
INTEGER refcount, * Anzahl der an das
                  Segment gebundenen
                  Prozesse
BOOLEAN res, * Systemresidenzindikator
INTEGER uid, * Benutzeridentifika-
            tion
INTEGER pid, * Prozessidentifikation
```

```

        INTEGER      backsaddr, * Adresse des Segments
                                im Hintergrundspei-
                                cher
        INTEGER      coreaddr  * Adresse des Segments
                                im Arbeitsspeicher
    );
                                * Segmentkontrollblock

segmenttab: VECTOR_OF sgb;

DECLARATIONS

                                * Segmentkontrollblöcke der Segmente
                                vom:
segmenttab shdata, * Typ 'sds' = { 'sdsw','sdsr' }
            shtext, * Typ 'scs'
            text,   * Typ 'cs'
            pown;   * Typ 'cds', 'ps', 'sts', 'ds'
INTEGER      addr;
INTEGER      i,j,
            pid;   * systemeindeutiger Prozessidentifi-
                    kation

segmenttyp segtyp;

PARAMETER
    INTEGER      nshdata, * Max. Anzahl der Segmente vom Typ
                        'sds'
            nshtext, * Max. Anzahl der Segmente vom Typ
                        'scs'
            ntext;   * Max. Anzahl der Segmente vom Typ
                        'cs'

EXTERNALREFS

    OFUN swap.swapin(segtyp,i);
    OFUN swap.swapout(segtyp,i);
    OFUN mem.shmfree(sharemap);
```

FUNCTIONS

OFUN shalloc(i);

* Parameter: i - Index auf ein Element im shdata-Vektor

* Effekt: Bindet einen Prozess an ein Segment des Typs 'sds'

EXCEPTIONS

i $\geq$ nshdata;

EFFECTS

TYPECASE shdata[i].refcount OF

≠0: TYPECASE shdata[i].loadcount OF

≠0: bind(sds,i,pid);

=0: bind(sds,i,pid);

 swapi(n,sds,i);

END

=0: create(sds,i);

 bind(sds,i,pid);

 swapi(n,sds,i);

END

OFUN shfree(i);

* Parameter: i - Index auf ein Element im shdata-Vektor

* Effekt: Löst die Bindung des laufenden Prozesses an das Segment

EXCEPTIONS

shdata[i] = UNDEFINED;

EFFECTS

release(sds,i);

```

TYPECASE shdata[i].loadcount OF
  =0: TYPECASE shdata[i].refcount OF
    ≥1: swapout(sds,i);
    =0: TYPECASE shdata[i].res OF
      TRUE: swapout(sds,i);
      FALSE: destroy(sds,i);
    END
  END
  ≠0: ;
END

```

```

OFUN shxalloc(i);
  * Parameter: i - Index auf ein Element im shtext-
    Vektor
  * Effekt: Bindet einen Prozess an ein Segment des
    Typs 'scs'

```

EXCEPTIONS

i ≥ nshtext

EFFECTS

```

TYPECASE shtext[i].refcount OF
  ≠0: TYPECASE shtext[i].loaccount OF
    ≠0: bind(scs,i,pid);
    =0: bind(scs,i,pid);
    swapin(scs,i);
  END
  =0: create(scs,i);
    bind(scs,i,pid);
    swapin(scs,i);
END

```

OFUN shxfree(i);

* Parameter: i - Index auf ein Element im shtext-Vektor

* Effekt: Löst die Bindung des laufenden Prozesses, an das Segment

EXCEPTIONS

shtext[i] = UNDEFINED;

EFFECTS

TYPECASE shtext[i].loadcount OF

=0: TYPECASE shtext[i].refcount OF

 <1: shmfree(sharemap);

 =0: destroy(scs,i);

END

≠0: ;

END

Bemerkungen:

Folgende Funktionen bleiben unverändert:

OFUN xalloc(i);

OFUN xfree(i);

VFUN currentscb(segtyp) → j;

VFUN getcoreaddr(segtyp) → addr;

Analog xalloc bzw. xfree sind folgende Funktionen spezifiziert:

OFUN xallocsh(i);

OFUN xfreesh(i);

Die Funktionen

OFUN release(seg,i);

OFUN bind(seg,i,pid);

OFUN destroy(seg,i);

werden aus vorhandenen Grundoperationen gebildet.

END_MODULE

5.3.8.2 MODULE Objektschutzsystem /obj /

TYPES

```

mode: {usermode, systemmode};

accessrights: {wr,      * write and read
               rd,      * read
               ex,      * execute
               np       * not present
               };      * Zugriffsrechte

segregister: STRUCT(
               accessright accessr, * Zugriffsrecht
               INTEGER      limit,  * Grenzregister
               reloc        * Basisregister
               );                * Segmentie-
                                   rungsregister

cap: STRUCT(
               segmenttyp segtyp,
               segregister segreg
               );

lcaplist: VECTOR_OF cap;
gcaplist: VECTOR_OF lcaplist;
lclist: VECTOR_OF segregister;

```

DECLARATIONS

```

gcaplist    clist;
lclist      sysclist,
            usrclist,
            macreg;

```

```

INTEGER      n,nt,nd,ns,nshw,nshr,nsht;
INTEGER      addr;
segmenttyp   segtyp;
mode         pmode;

```

PARAMETER

```

INTEGER maxseg,      * max. Anzahl der prozessspezifischen
                      Segmente
                      maxreg,      * max. Anzahl der Reg. in der Segmenttab.
                      maxlmem,     * max. Benutzerspeicher im lok. Speicher
                      maxshmem;    * max. Benutzerspeicher im gem. Speicher
accessrights wrkey,  * Zugriffsschlüssel
                      rdkey,
                      exkey,
                      npkey;

```

DEFINITIONS

```

INTEGER nseg(n) IS { ENTIRE ((n + 256) / 256 / 256) };
                      * Liefert die Anzahl der realen Segmente
                      eines Ladeobjektsegments

```

```

sizedseg(n) IS { TYPECASE nseg(n) OF
                  _1: 256*256;
                  n=n'-(256*256);
                  =0: ENTIRE ((n+256)/256))

```

END

};

```

* Zerlegt die Ladeobjektsegmente in phys.
  Segmente und liefert die Werte für die
  Grenzregister

```

EXTERNALREFS

```

seg.segmenttyp;
VFUN proc.m.gcpce → i;
VFUN seg.getcoreaddr(segtyp) → addr;

```

FUNCTIONS

OFUN estabur(nt,nd,ns,nshw,nshr,nsht,j);

- * Parameter: nt - Grösse des Codesegments
- nd - Grösse des Datensegments
- ns - Grösse des Stacksegments
- nshw - Grösse des 'Shared' Datensegments - w/r
- nshr - Grösse des 'Shared' Datensegments - r
- nsht - Grösse des 'Shared' Codesegments
- j - clist - Index

- * Effekt: Initialisiert die prozessspezifische (lokale) Zugriffsliste

EXCEPTIONS

nseg(nt)+nseg(nd)+nseg(ns)+nseg(nshw)+nseg(nshr)+nseg(nsht)>15;

nt+nd+ns > maxlmem;

nshw+nshr+nsht > maxshmem;

EFFECTS

$\forall i \mid 0 \leq i \leq (\text{maxseg} - 1): \text{access}(j,i);$

OFUN sureg(j);

- * Parameter: j - clist-Index

- * Effekt: Lädt die durch j spezifizierte lokale Zugriffsliste in den Zugriffslistenpuffer des Betriebssystemkerns und trägt die Basisadresse jedes definierten Segments in das entsprechende Basisregister ein

EFFECTS

sysclist = clist[j] ^

sysclistfil.reloc = getcoreaddr(clist[j].segtype);

$\forall i \mid 0 \leq i \leq (\text{maxseg} - 1);$

```
OFUN addr(pmode);
    * Parameter: pmode - Systemmodus/Benutzermodus
    * Effekt: Lädt beim 'Schutzumgebungswechsel'
              die Segmenttabelle mit der entsprechenden
              Zugriffsliste aus dem Zugriffslistenpuffer
              des Betriebssystems
```

EFFECTS

```
    IF usermode
        THEN macreg = sysclist;
    IF systemmode
        THEN macreg = usrclist;
```

```
OFUN access(j,i);
    * Parameter: i -Segmenttabellenregisterindex
                  j -clist - Index
    * Effekt: Lädt die durch i spezifizierten
              Segmenttabellenregister ( Schutzschlüsselregister,
              Grenzregister ) der lokalen Zugriffsliste
```

HIDDEN;

EFFECTS

```
    TYPECASE clist[j,i].segtyp OF
    cds:      clist[j,i].accessr= wrkey;
              clist[j,i].limit  = sizeseg(nd);
    cs:       clist[j,i].accessr= exkey;
              clist[j,i].limit  = sizeseg(nt);
    ds:       clist[j,i].accessr= wrkey;
              clist[j,i].limit  = sizeseg(nd);
    scs:      clist[j,i].accessr= exkey;
              clist[j,i].limit  = sizeseg(nsht);
    sds:      clist[j,i].accessr= wrkey;
```

```
        clist[j,i].limit  = sizeseg(nshw);  
sdsr:   clist[j,i].accessr= rdkey;  
        clist[j,i].limit  = sizeseg(nshr);  
ps:     clist[j,i].accessr= wrkey;  
        clist[j,i].limit  = sizeseg(nd);  
sts:    clist[j,i].accessr=wrkey;  
        clist[j,i].limit  = sizeseg(ns);  
default: clist[j,i].accessr= npkey;  
        clist[j,i].limit  = sizeseg(0);  
END
```

END_MODULE

5.3.8.3 Spezifikation

MODULE lader /load/

TYPES

```

header:      STRUCT(
                proctype      proctyp, * Ladeobjekt-
                                typ
                INTEGER        lc,      * Länge des:
                                Codesegments
                INTEGER        ldsdw,   * gem. Datensegm.
                                (read/write)
                INTEGER        ldsdr,   * gem. Datensegm.
                                (read only)
                INTEGER        lds,     * lok. Datensegm.
                INTEGER        lsts,    * Stacksegments
                INTEGER        lps      * Datensegm. des
                                Prozesstyps
                                'prog'
                                * Ladeobjekt-
                                kopfinformation
            );
    
```

DECLARATIONS

```

header      lkb;                * Laderkontrollblock
proctype     processtyp;        * Prozesstyp
puids       puid;              * Prozessoridentifikation
INTEGER     i,j;               * Index
segmenttyp  segtyp;            * Segmenttyp
    
```

PARAMETERS

VEC_OF_INT	maxmem;	* Max. Grösse der lok. Speicher
INTEGER	maxshmen;	* Max. Grösse des gem. Speichers
INTEGER	lsysds;	* Länge der 'System Data Segments'
INTEGER	maxproc;	* Max. Anzahl der Prozessoren
INTEGER	maxnseg;	* Max. Anzahl der Segmente eines Prozesses

DEFINITIONS

```
INTEGER nseg(n) IS ENTIRE ((n + 256/256)/256);  
      * Liefert die Anzahl der realen Segmente  
      eines Ladeobjektsegments
```

EXTERNALREFS

```
proc.m.proctype;  
proc.m.puids;  
seg.segmenttyp;  
VFUN proc.m.currenttype → processtyp;  
VFUN proc.m.currentproc → puid;  
VFUN proc.m.currentfathertype → processtyp;  
VFUN proc.m.currentpcb → j;  
OFUN seg.xfree(i);  
OFUN seg.xfreesh(i);  
OFUN seg.shxfree(i);  
OFUN seg.xalloc(i);  
OFUN seg.xallocsh(i);  
OFUN seg.shxalloc(i);  
OFUN seg.shalloc(i);  
OFUN seg.shfree(i);  
VFUN seg.currentscb(segtyp) → i;  
OFUN obj.estabur(nt,nd,ns,nshw,nshr,nsht);  
OFUN obj.sureg(j);
```

FUNCTIONS

OFUN loader(filename);

* Parameter: filename - Dateiname des Ladeobjekts

* Effekt: Lädt das Ladeobjekt gemäss Steueranweisungen im Laderkontrollblock

EXCEPTIONS

lkb.proctyp $\neq$ currenttype;

TYPECASE currenttype OF

procl: lkb.lc+lkb.lds+lkb.lsts+lsysds >
maxmem(currentproc);
nseg(lkb.lc)+nseg(lkb.lds)+nseg(lkb.lsts)+
nseg(lsysds) > maxnseg;

procg: lkb.lc+lkb.lds+lkb.lsts+lsysds >
maxmem(currentproc);
lkb.lsdsw+lkb.lsdsw > maxshmem;
nseg(lkb.lc)+nseg(lkb.lds)+nseg(lkb.lsts)+
nseg(lsysds)+nseg(lkb.lsdsw)+nseg(lkb.lsdsw) >
maxnseg;

procg: lkb.lc+lkb.lsdsw+lkb.lsdsw+lkb.lsts+
lkb.lps+lsysds > maxshmem;
nseg(lkb.lc)+nseg(lkb.lsdsw)+nseg(lkb.lsdsw)+
nseg(lkb.lsts)+nseg(lkb.lps)+nseg(lsysds) >
maxnseg;

END

EFFECTS

getheader;

TYPECASE currentfathertype OF

procl: xfree(currentscb(cs));
expand(lsysds);

procg: xfreesh(currentscb(cs));
shfree(currentscb(sds));
expand(lsysds);

```
procg:  shxfree(currentscb(scs));
        shfree(currentscb(sds));
        shexpand(1sysds);

END

TYPECASE currenttype OF
proc1:  xalloc(currentscb(cs));
        expand(1kb.lds+1kb.1sts+1sysds);
        getsegment;

procgt: xallocsh(currentscb(cs));
        shalloc(currentscb(sds));
        expand(1kb.lds+1kb.1sts+1sysds);
        getsegment;

procg:  shxalloc(currentscb(scs));
        shalloc(currentscb(sds));
        shexpand(1kb.1ps+1kb.1sts+1sysds);
        getsegment;

END

IF currenttype = procg
  THEN estabur(0,1kb.1ps,1kb.1sts,1kb.1sdsw,
               1kb.1sdsr,1kb.1c);
  ELSE estabur(1kb.1c,1kb.1ds,1kb.1sts,1kb.1sdsw,
               1kb.1sdsr,0);

sureg(currentpcb);
initstack;
initregs;
```

Funktionen, die vom Monoprozessorsystem weitgehend unmodifiziert übernommen werden konnten, bzw. deren Implementierung analog zu bestehenden Funktionen erfolgen kann:

(EXTERNALREFS)

```
OFUN io.getheader;
    * Effekt:   Liest den Laderkontrollblock

OFUN io.getsegment;
    * Effekt:   Liest Segmente des Ladeobjekts
```

OFUN proc.m.initstack;

* Effekt: Initialisiert Stacksegment

OFUN proc.m.initregs;

* Effekt: Initialisiert Registerpuffer
Löscht 'Softwareinterrupts'

OFUN proc.m.update;

* Effekt: Aktualisiert Prozesskontrollblock

(FUNCTIONS)

OFUN expand(newsize);

* Effekt: Beschafft neuen Arbeitsspeicher für
'Process Own Segments'
Falls: newsize < oldsize:
Speicherfreigabe
Falls: oldsize < newsize:
Speicheranforderung
Falls: genügend Arbeitsspeicher
vorhanden:
Transfer: 'Process Own Segment'
in den neuen Speicherbereich
Freigabe des alten Speichers
Falls: nicht genügend Arbeitsspeicher
vorhanden:
Prozess wird ausgelagert

OFUN shexpand(newsize);

* Effekt: Analog expand, jedoch nur für
'Process Own Segment' der Prozess-
klasse 'procg'

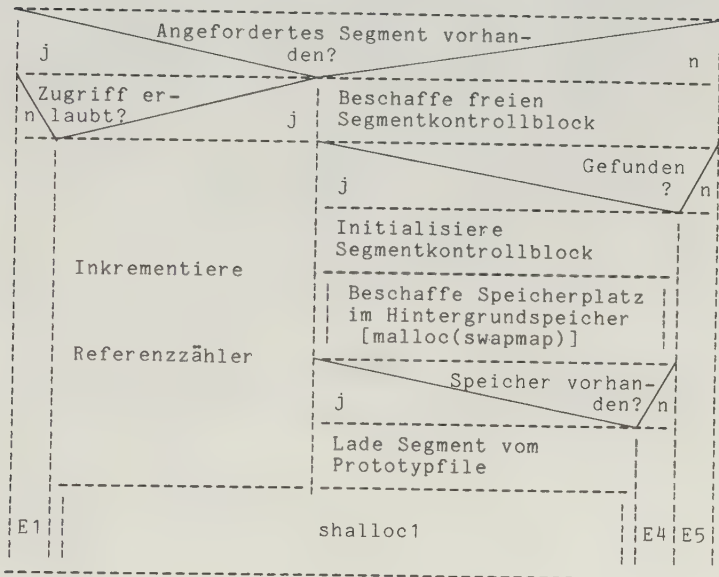
END_MODULE

5.3.9 Implementierung

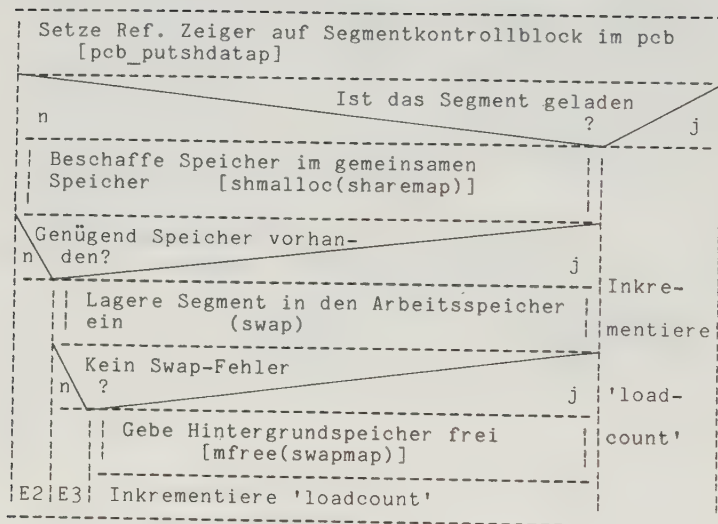
5.3.9.1 Segmentverwaltung

shalloc(ip)

Parameter: ip - Zeiger auf Segmentkontrollblock



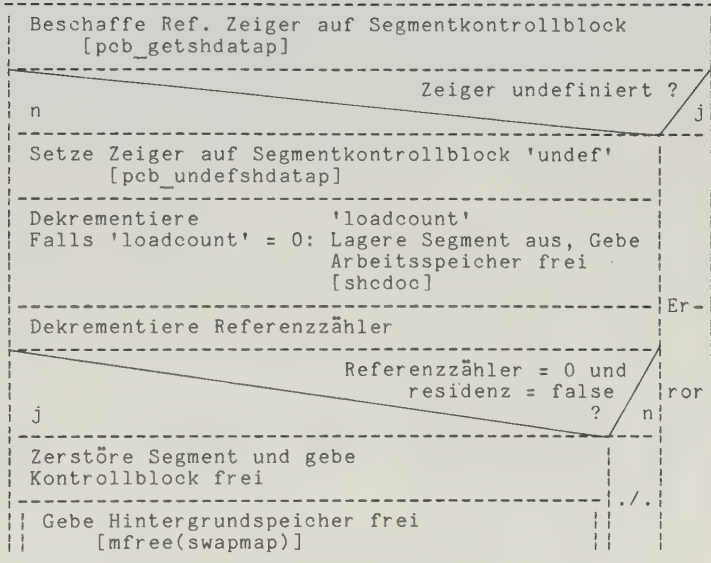
shalloc1



E1: panic("no accessrights")
 E2: panic("no shared memory")
 E3: panic("swap error")
 E4: panic("out of swapspace")
 E5: panic("out of shdata")

shfree(ip)

Parameter: ip - Zeiger auf Segmentkontrollblock



Weitgehend analog:

shxalloc

shxfree

sureg()

```
-----  
| Beschaffe aus den Prozesskontrollblöcken die An- |  
| fangsadressen der an den Prozess gebundenen Segmente |  
-----  
| Lade aus der Zugriffsliste des Prozesses den Benut- |  
| zerzugriffslistenpuffer des Betriebssystemkerns und |  
| lade die Basisregisterfelder mit den Basisadressen |  
| der entsprechenden Segmente |  
-----
```

addr()

nicht geändert

5.3.9.3 Lader

loader(filename)

Parameter: filename-Dateiname des Ladeobjekts

Lese Laderkontrollblock			
j	Entspricht Prozesstyp dem Ladeobjekttyp?		n
j	Anzahl und Grösse der Segmente korrekt?		n
CASE Prozesstyp des Vater-(erzeugenden)Proz.			
procl	procgt	procg	
Löse Bindung Prozess - Codesegment ggf. mit Speicherfreigabe			
(xfree)	(xfreesh)	(shxfree)	
Löse Bindung Prozess - Datensegment Typ 'sds'			
./.	(shfree)	(shfree)	
Löse Bindung Prozess - Datensegment/Stacksegment mit Speicherfreigabe			
(expand)	(expand)	(shexpand)	
CASE Prozesstyp des ladenden (current) Proz.			
procl	procgt	procg	
Binde Prozess - Codesegment, ggf. mit Speicheranforderung			
(xalloc)	(xallocsh)	(shxalloc)	
Binde Prozess - Datensegment des Typs 'sds', ggf. mit Speicheranforderung			
./.	(shalloc)	(shalloc)	

Fortsetzung: loader

Beschaffe zusammenhängenden Speicherbereich für das 'Process Own Segment' und kopiere, falls notwendig, das 'System Data Segment'		
(expand)	(expand)	(shexpand)
Lade Datensegment		
Initialisiere Stacksegment		
Generiere Zugriffslistenprototyp im 'System Data Segment'		
(estabur)		
Lade Benutzerzugriffslistenpuffer im Betriebssystemkern		
(sureg)		
Ergänze Prozesskontrollblockdaten		
Lösche Softwareinterrupts und Maschinen- registerpuffer		

5.4 Freispeicherverwaltungssystem

Vom Freispeicherverwaltungssystem werden die Speicherobjekte 'local memory', 'shared memory' und 'swap area' verwaltet.

Da alle Speicherobjekte die gleiche Struktur besitzen - lediglich die Verwaltungseinheiten unterscheiden sich - liegt es nahe, das gesamte System zu einem Modul zusammenzufassen.

Vom Monoprozessorsystem konnte für die Objekte 'local memory' und 'swap area' der Modul nahezu unverändert übernommen werden. Zur Verwaltung des gemeinsamen Speichers mußten zusätzlich Funktionen ('shmfree', 'shmalloc') implementiert werden, die jedoch dem gleichen Algorithmus gehorchen, wie die Funktionen des restlichen Freispeichersystems.

Desweiteren mußte die Initialisierungsfunktion neu gefaßt werden. Dadurch ist die Voraussetzung geschaffen worden, das Betriebssystem nicht mehr in einem zusammenhängenden Speicherbereich halten zu müssen (local part/shared part). Außerdem kann ein lückenhafter Adreßraum des Arbeitsspeichers verwaltet werden.

5.4.1 Verwaltungsstrategie

Der Speicherplatz wird in zusammenhängenden Blöcken in Einheiten von 256 Byte in den Arbeitsspeichern und in Einheiten von 512 Byte in den Hintergrundspeichern nach der Strategie 'First Fit' vergeben. Bei der Rückgabe von Speicher werden benachbarte Speicherbereiche verschmolzen.

5.4.2 Spezifikation

MODULE freispeicherverwaltung / mem /

TYPES

```
map: STRUCT(  
    INTEGER size, * Anzahl der Speicherblöcke  
    INTEGER addr * Basisadresse  
);  
maps: VECTOR_OF map;
```

DECLARATIONS

```
maps    sharemap, * gemeinsamer Speicher  
        coremap,  * privater Speicher  
        swapmap;  * Hintergrundspeicher  
INTEGER i,j,x;  
INTEGER size,addr;
```

PARAMETERS

```
        * Max. Anzahl der zusammenhängenden  
        Speicherbereiche:  
INTEGER shmaphsize, * im gemeinsamen Speicher  
        cmaphsize,  * im privaten Speicher  
        swapsize,   * im Hintergrundspeicher  
userbot,  * kleinste phys. Adresse im Benutzer-  
          adressraum des lokalen Speichers  
usertop,  * grösste phys. Adresse im Benutzer-  
          adressraum des lokalen Speichers  
shuserbot, * kleinste phys. Adresse im Benutzer-  
          adressraum des gemeinsamen Speichers  
shusertop, * grösste phys. Adresse im Benutzer-  
          adressraum des gemeinsamen Speichers
```

swapbot, * 0
 swaptop; * grösste phys. Adresse im Hintergrund-
 speicher

DEFINITIONS

INTEGER shgblock IS { sharemap[i] | ($\neg \exists$ (sharemap[j] |
 sharemap[j].size > sharemap[i].size) ,
 $1 \leq i, j \leq \text{shmapsize}$)};
 INTEGER cgblock IS { coremap[i] | ($\neg \exists$ (coremap[j] |
 coremap[j].size > coremap[i].size) ,
 $1 \leq i, j \leq \text{cmapsize}$)};
 INTEGER swapgblock IS { swapmap[i] | ($\neg \exists$ (swapmap[j] |
 swapmap[j].size > swapmap[i].size) ,
 $1 \leq i, j \leq \text{swapsize}$)};

FUNCTIONS

OVFUN shmalloc(size) $\longrightarrow$ addr;
 * Parameter: size - Grösse des angeforderten Speicherbereichs
 addr - Basisadresse des Speicherbereichs
 * Effekt: Speicherbereich der gewünschten Grösse wird, falls vorhanden, zugeteilt

EXCEPTIONS

shgblock < size;

EFFECTS

addr = sharemap[i].addr,
 i | ($\neg \exists$ sharemap[j] | (sharemap[j].size > size $\wedge$
 $j < i$, $1 \leq i, j \leq \text{shmapsize}$);

OFUN shmfree(size,aa);

- * Parameter: size - Grösse des Speicherbereichs
aa - Adresse des Speicherbereichs
- * Effekt: Ordnet den freigegeb. Speicherbereich in die Freispeicherliste ein und verschmilzt, falls vorhanden, benachbarte Speicherbereiche

DEFINITIONS

BOOLEAN B1(i) IS { $\exists$ sharemap[i] | sharemap[i].addr +
sharemap[i].size = aa ,
 $1 \leq i \leq \text{shmapsize}$ };

BOOLEAN B2(j) IS { $\exists$ sharemap[j] | sharemap[j].addr =
aa + size , $1 \leq j \leq \text{shmapsize}$ };

EXCEPTIONS

$\exists$ sharemap[i] | sharemap[i].addr $\leq$ aa
< (sharemap[i].addr + sharemap[i].size),
 $1 \leq i \leq \text{shmapsize}$;

EFFECTS

TYPECASE B1(i) OF

TRUE: TYPECASE B2(j) OF

TRUE: sharemap[i].size = sharemap[i].size' + size +
sharemap[j].size $\wedge$
sharemap[j] = UNDEFINED;

FALSE: sharemap[i].size =
sharemap[i].size' + size;

END

FALSE: TYPECASE B2(j) OF

TRUE: sharemap[j].addr = aa $\wedge$
sharemap[j].size =
sharemap[j].size' + size;

FALSE: sharemap[x].addr = aa $\wedge$
sharemap[x].size = size $\wedge$
sharemap[x]' = UNDEFINED;

END

END

OVFUN malloc(mp,size) → addr;

* Parameter: mp - swapmap oder coremap
size - Grösse des angeforderten Speicherbereichs

* Effekt: Speicherbereich der gewünschten Grösse wird, falls vorhanden, zugeteilt

EXCEPTIONS, EFFECTS analog shmalloc;

OFUN mfree(mp,size,aa);

* Parameter: mp - swapmap oder coremap
size - Grösse des zurückgegebenen Speicherbereichs
aa - Adresse des Speicherbereichs

* Effekt: Ordnet den zurückgegebenen Speicherbereich in die entsprechende Freispeicherliste ein und verschmilzt, falls vorhanden, benachbarte Speicherbereiche

EXCEPTIONS, EFFECTS analog shmfree;

OFUN initfreestorage;

* Effekt: Initialisiert die Freispeicherlisten

EXCEPTIONS

none;

EFFECTS

sharemap[1].addr = shuserbot;
sharemap[1].size = shusertop - shuserbot;
coremap[1].addr = userbot;
coremap[1].size = usertop - userbot;
swapmap[1].addr = swapbot;
swapmap[1].size = swaptop - swapbot;

5.4.3 Implementierung

shmalloc(size)

Parameter: size - Grösse des angeforderten Speicherbereichs

Suche den ersten freien Speicherbereich, der grösser oder gleich dem angeforderten Bereich ist	
j	Gefunden ? n
Trage den gewünschten Speicherbereich aus der Freispeicherliste aus	re-
return(Basisadresse des Speicherbereichs)	turn
	(0)

shmfree(size,aa)

Parameter: size - Grösse des zurückgegebenen Speicherbereichs
aa - Basisadresse des Speicherbereichs

Zurückgegebener Speicher Teil des Freispeichers	
j	? n
Füge den zurückgegebenen Speicherbereich in die Freispeicherliste ein und verschmelze, falls möglich, benachbarte Speicherbereiche	Er-
	ror

5.5 Koordinierungssystem

Das Koordinierungssystem ist (gemäß Courtois-Relation) /Hofm77/ in drei Ebenen hierarchisch gegliedert.

Auf der niedrigsten Ebene ist ein 'Lock/Unlock'-Mechanismus zur Sicherstellung des gegenseitigen Ausschlusses bei kurzen kritischen Abschnitten implementiert.

Auf der nächsten Ebene ist mit dem PV-System für Systemprozesse der 'klassische Semaphor' realisiert. Auf eine dynamische Speicherverwaltung ist auf dieser Ebene verzichtet worden - sämtliche Semaphore werden beim Systemgenerieren erzeugt und beim Systemstart initialisiert.

Dem Benutzer steht ein Koordinierungsmechanismus vom Typ PV_{chunk} zur Verfügung. Der direkte Zugriff auf die Koordinierungsvariablen ist dem Benutzer verwehrt. Er kann nur über die vom Modul 'sema' zur Verfügung gestellten Funktionen 'crsem', 'dstsem', 'delsem', 'psem' und 'vsem' seine Koordinierungsvariablen manipulieren.

Die dynamische Bindung der Semaphorelemente an die Benutzerprozesse wird vom Betriebssystem organisiert. Ein vom Benutzer beeinflussbarer Zeitüberwachungsmechanismus (time-out) löst Benutzersystemverklemmungen auf und terminiert die beteiligten Prozesse.

5.5.1 Koordinierungssystem für Systemprozesse

5.5.1.1 Lock/Unlock-Mechanismus

Der "Lock/Unlock"-Mechanismus (binärer Semaphor) ist aus Effektivitätsgründen in Assemblersprache realisiert. Der Zugriff auf die Lockvariable wird mit dem 'Test and Set'-Befehl koordiniert. In einer 'lock/unlock'-Klammer ist eine Unterbrechung durch externe Ereignisse (Interrupts) nicht möglich.

5.5.1.2 PV-System für Systemprozesse

Im PV-System für Systemprozesse ist der 'klassische Semaphor' für die Koordinierung unterhalb der Betriebssystemschnittstelle implementiert.

Folgende Funktionen stellt der Modul zur Verfügung:

syssemit: Während der Initialisierungsphase des Betriebssystems werden durch den Aufruf dieser Funktion sämtliche Systemsemaphore vorbesetzt.

syssemv(sn,tr): 'signal'-Operation auf den Semaphor 'sn'.

syssemp(sn,tr): 'wait'-Operation auf den Semaphor 'sn'.
'tr' = true: 'Traceoption' für Testzwecke

5.5.1.3 Spezifikation

MODULE lock/unlock / lock /

DECLARATIONS

INTEGER lockvar;

FUNCTIONS

OFUN lock($\uparrow$ lockvar);

* Parameter: $\uparrow$ lockvar - Adresse der Lockvariablen

* Effekt: $\langle$ lockvar $\rangle = 1$; Interruptsperre gesetzt

EXCEPTIONS

none;

EFFECTS

lockvar = 1 $\wedge$ Interrupts disable;

OFUN unlock($\uparrow$ lockvar);

* Parameter: $\uparrow$ lockvar - Adresse der Lockvariablen

* Effekt: $\langle$ lockvar $\rangle = 0$; Interruptsperre zurück-
gesetzt

EXCEPTIONS

none;

EFFECTS

lockvar = 0 $\wedge$ Interrupts enable;

OFUN unlockof(↑lockvar);

* Parameter: ↑lockvar - Adresse der Lockvariablen

* Effekt: <lockvar> = 0;

EXCEPTIONS

none;

EFFECTS

lockvar = 0;

END_MODULE

MODULE systemsemaphor / syssem /

TYPES

```
queuep:  DESIGNATOR;
bl:      STRUCT(
            puids puid,      * Prozessoridentifikation
            masks mask      * Prozessormasken
        );
sema:    STRUCT(
            INTEGER syvalue, * Wert
            queuep syfirst, * Zeiger auf das 1. belegte
                           Warteschlangenelement
            queuep syfqu,   * Zeiger auf das 1. freie
                           Warteschlangenelement
            VEC_OF bl block * Warteschlange
        );                * Semaphor
```

DECLARATIONS

```
SET_OF sema syssem;
puids      puid;
messagecode messcode;
message     mess;
```

PARAMETER

```
INTEGER maxbl;      * Max. Länge der Warteschlange
INTEGER semprio;    * Priorität
INTEGER messcode;   * Nachrichtencode
```

EXTERNALREFS

```
mess.message;
mess.messagecode;
procm.puids;
procm.masks;
OFUN procm.wakeup(chan);
OFUN procm.sleep(chan,prio);
```

```
VFUN procn.currentproc → puid;
OFUN mess.putmess(puid,messcode,↑mess);
```

FUNCTIONS

```
OFUN syssemit;
    * Effekt: Initialisierung der Systemsemaphore
```

EXCEPTIONS

```
none;
```

EFFECTS

```
∀ sema ∈ sysem:
    semai.syvalue = 1 ∧
    sysem.syfirst = UNDEFINED ∧
    semai.syfqu = ↑ block[0] ∧
    semai.block[j] = 0, ∀ j | 0 ≤ j < maxbl;
```

```
OFUN syssemv(i);
```

```
    * Parameter: i      - EXTRACTOR des Semaphors
    * Effekt: Signaloperation
```

EXCEPTIONS

```
semai.syfirst = UNDEFINED ∧ semai.syvalue ≤ -1;
```

EFFECTS

```
semai.syvalue = semai.syvalue' + 1;
TYPECASE semai.syvalue OF
    <1: TYPECASE semai.syfirst.puid = currentproc OF
        TRUE: wakeup(semai.syfirst);
        TYPECASE semai.syvalue OF
            =0: semai.syfirst=UNDEFINED;
            <0: semai.syfirst=semai.syfirst'
                + 1 MOD (maxbl - 1);
```

```
END
```

```

FALSE: semai.syvalue=semai.syvalue';
      putmess(semai.syfirst.puid,
              messcode,↑mess);

      END
    ≥1: ;
  END

OFUN syssemp(i);
  * Parameter: i          - EXTRACTOR des Semaphors
  * Effekt: Waitoperation

EXCEPTIONS
  semai.value = -maxbl;

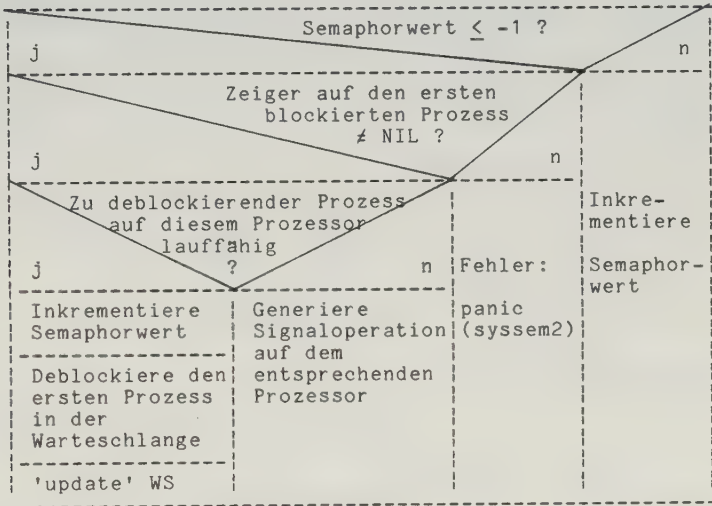
EFFECTS
  semai.syvalue = semai.syvalue' - 1;
  TYPECASE semai.syvalue OF
    <0: sleep(semai.syfqu,semprio);
      semai.syfqu = semai.syfqu' + 1 MOD (maxbl-1);
      TYPECASE semai.syfirst OF
        UNDEF: semai.syfirst = semai.syfqu';
        ↯UNDEF: ;
      END
    ≥0: ;
  END
END

END_MODULE

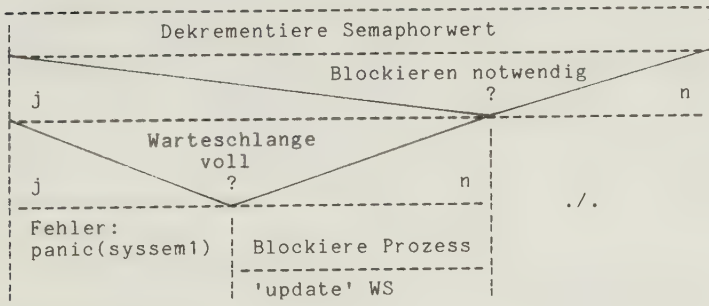
```

5.5.1.4 Implementierung

syssemv(sn,tr) sn: Zeiger auf Semaphor
 tr: Traceindikator =0 -- kein Trace
 =1 -- Trace



syssemp(sn,tr) sn: Zeiger auf Semaphor
 tr: Traceindikator =0 -- kein Trace
 =1 -- Trace



5.5.2 Koordinierungssystem für Benutzerprozesse

Hiermit wird dem Benutzer ein Koordinierungssystem vom Typ PV_{chunk} zur Verfügung gestellt. Die vom Benutzer erzeugten Semaphore gehören zur Umgebung eines Prozeßsystems und sind daher für diesen Bereich eindeutig zu vereinbaren.

Als Betriebssystemdienste sind folgende Funktionen zugänglich:

`crsem(sname,initval,tout)`: Es wird ein Semaphore mit Namen 'sname' und dem Anfangswert 'initval' eingerichtet. Die Zeitüberwachungskonstante für diesen Semaphore ist 'tout'.

`vsem(sname,ch)`: 'signal'-Operation auf den Semaphore 'sname' mit dem 'Chunk'-Wert 'ch'.

`psem(sname,ch)`: 'wait'-Operation auf den Semaphore 'sname' mit dem 'Chunk'-Wert 'ch'.

`dstsem(sname)`: Löschen des Semaphors 'sname'.

`delsem(userid)`: Anstoßen der Löschoperation für alle Semaphore, die für den Benutzeridentifikator 'userid' eingerichtet sind. Die Löschung erfolgt, sobald keine Referenzen mehr auf irgendeinen der zu löschenden Semaphore besteht.


```

        VEC_OF bl blk,      * Warteschlange
        INTEGER timeout,    * Zeitüberwachungskon-
                               stante
    );                        * Semaphorelement

```

DECLARATIONS

```

    VECTOR_OF semahead head;
    SET_OF sema      usema;
    INTEGER          i,j;
    INTEGER          uid,userid;
    puids            puid;
    messagecode      messcode;
    message          mess;

```

PARAMETERS

```

    INTEGER semmax,      * Max. Anzahl an Semaphorelementen
    sempri,              * Priorität des an einem Semaphor
                          blockierten Prozesses
    uidmax,              * Max. Anzahl verschiedener Benutzer
    maxbl,               * Max. Länge der Warteschlange
    semauid:              { semai | semai.uid = uid,
                          0 ≤ i < semmax };
                          * Menge der Sem. des Ben. 'uid'
    semafree:            { semai | semai.uid = UNDEFINED,
                          0 ≤ i < semmax };
                          * Menge der freien Semaphoren
    headaktiv:          { head[j] | head[j].uid ≠ UNDEFINED,
                          0 ≤ j < uidmax };
                          * Semaphoranker

```

DEFINITIONS

```

    sema semasname      IS {semai | semai.name = sname ,
                          0 ≤ i < semmax };
    semahead headuid    IS {head[j] | head[j].uid = uid ,
                          0 ≤ j < uidmax };

```

EXTERNALREFS

```
mess.message;
mess.messagecode;
procm.puids;
procm.masks;
OFUN procm.wakeup(chan);
OFUN procm.sleep(chan,prio)
VFUN procm.currentproc  $\rightarrow$  puid;
OFUN mess.putmess(puid,messcode, $\uparrow$ mess);
```

FUNCTIONS

```
OFUN seminit;
```

* Effekt: Initialisierung von 'usema' und 'head'

EXCEPTIONS

```
none;
```

EFFECTS

```
 $\forall$  sema  $\in$  usema:
```

```
semai.uid = UNDEFINED  $\wedge$   
semai.name = UNDEFINED  $\wedge$   
  
semai.value = 0  $\wedge$   
semai.first = UNDEFINED  $\wedge$   
semai.fqu =  $\uparrow$  blk[0]  $\wedge$   
semai.timeout = 0;
```

```
 $\forall$  head[j] ,  $0 \leq j < \text{uidmax}$ :
```

```
head[j].uid = UNDEFINED  $\wedge$   
head[j].pointer = UNDEFINED  $\wedge$   
head[j].counter = 0  $\wedge$   
head[j].delmark = FALSE;
```

OFUN vsem(sname,ch);

- * Parameter: sname - Name des Semaphors
 ch - Chunk-Wert
- * Effekt: Signaloperation

Spezifikation analog syssem.syssemv

OFUN psem(sname,ch);

- * Parameter: sname - Name des Semaphors
 ch - Chunk-Wert
- * Effekt: Waitoperation

Spezifikation analog syssem.syssemp;

OFUN crsem(sname,initval,tout);

- * Parameter: sname - Name des zu kreierenden Semaphors
 initval - Anfangswert
 tout - Zeitüberwachungskonstante
- * Effekt: Erzeugt für den aufrufenden Benutzer einen
Semaphor gemäss Parameter

DEFINITION

EXTRACTOR k IS {i | sema_i.uid = UNDEFINED ,
 0 ≤ i < semmax };

EXCEPTIONS

sema_{sname} ∈ sema_{uid};
head[uid].delmark = TRUE;

EFFECTS

```

TYPECASE semauid = ∅ OF
  TRUE: headuid = head[uid] ∧
        |headaktiv| = |headaktiv|' + 1;
  FALSE: ;
semasname = semak | semak.name = sname ∧
               semak.uid = uid ∧
               semak.value = initval ∧
               semak.timeout = tout ) ∧
               |semafree| = |semafree|' - 1 ∧
               semauid = semauid' UNION semasname;

```

OFUN dstsem(sname);

- * Parameter: sname - Name des Semaphors
- * Effekt: Löscht den Semaphor

EXCEPTIONS

```

semasname = ∅ semauid;
semasname.first = ∅UNDEFINED;
head[uid].delmark = TRUE;

```

EFFECTS

```

semasname = semauid \ semasname ∧
semafree = semafree' UNION semasname;
IF semauid = ∅
  head = head' \ headuid;

```

```
OFUN delsem(userid);
```

```
  * Parameter: userid - Benutzeridentifikation
```

```
  * Effekt: Leitet Löschung aller Semaphore des  
            angegebenen Benutzers ein
```

```
EFFECT
```

```
  head[userid].delmark = TRUE;
```

```
  TYPECASE head[userid].counter  $\neq$  0 OF
```

```
    TRUE: delay(counter = 0);
```

```
    FALSE: ;
```

```
   $\forall$  sema  $\in$  semauserid:
```

```
    seami.uid = UNDEFINED  $\wedge$ 
```

```
    semai.name = UNDEFINED  $\wedge$ 
```

```
    seami.value = 0  $\wedge$ 
```

```
    semai.first = UNDEFINED  $\wedge$ 
```

```
    semai.fqu =  $\uparrow$  blk[0]  $\wedge$ 
```

```
    semai.timeout = 0;
```

```
  head[userid]:
```

```
    head[userid].uid = UNDEFINED  $\wedge$ 
```

```
    head[userid].pointer = UNDEFINED  $\wedge$ 
```

```
    head[userid] = 0  $\wedge$ 
```

```
    head[userid] = FALSE;
```

```
  semafree = semafree ' UNION semauid';
```

```
  head = head* \ headuid;
```

```
END_MODULE
```

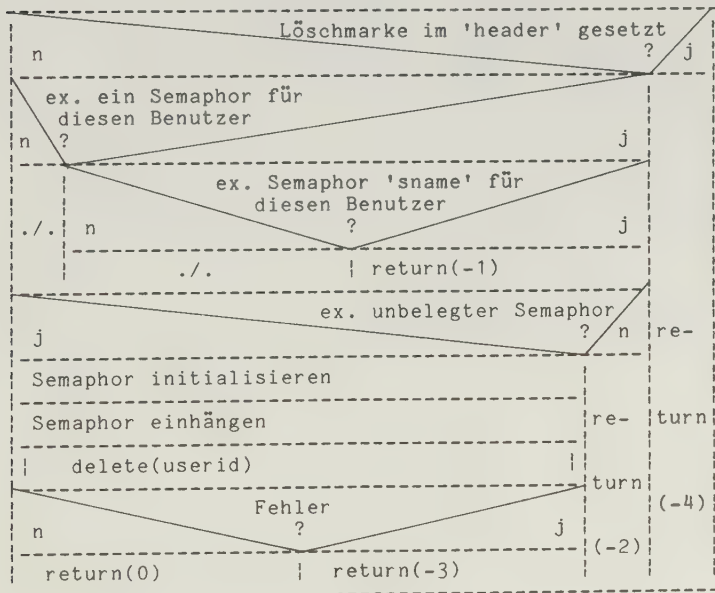
5.5.2.2 Implementierung

Implementierungshinweise:

- Die an einem Benutzerprozesssystem gebundenen Koordinierungsvariablen werden über einen benutzerspezifischen Verwaltungskopf ('header') in einer Baumstruktur verwaltet. Nicht gebundene Koordinierungsvariable werden in einem 'pool' gehalten.
- Die Abarbeitung der Warteschlange wird nach der Strategie FIFO vorgenommen.

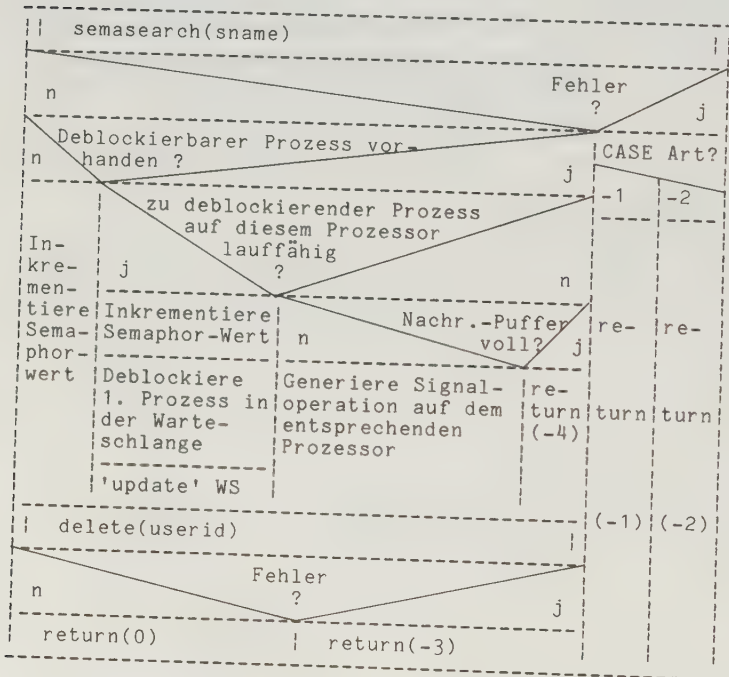
crsem(sname,initval,tout)

Parameter: sname - Name des zu erzeugenden Semaphors
 initval - Anfangswert des zu erzeug. Semaphors
 tout - Zeitüberwachungskonstante für den zu erzeugenden Semaphor



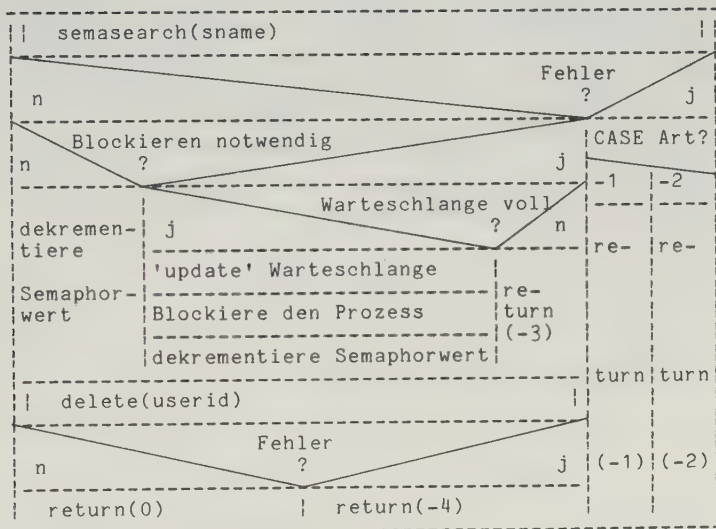
vsem(sname,ch)

Parameter: sname - Name des Semaphors, auf dem die Signal-
operation ausgeführt werden soll
ch - Chunk-Wert



psem(sname,ch)

Parameter: sname - Name des Semaphors, auf den die Wait-
operation auszuführen ist
ch - Chunk-Wert



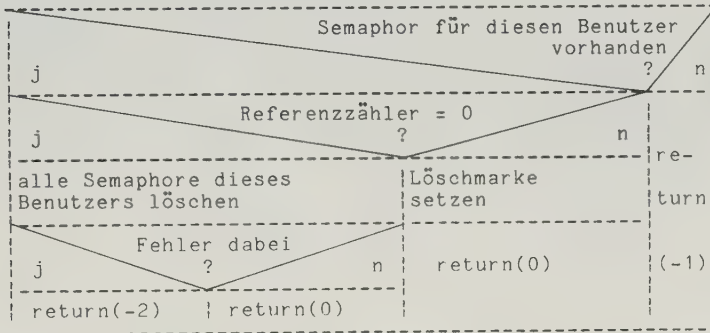
dstsem(sname)

Parameter: sname - Name des zu löschenden Semaphors

semasearch(sname)			
n	Fehler ?	j	
n	Prozess an 'sname' blockiert?	j	CASE Art?
Semaphor 'sname' ausketten		-1	-2
Semaphor 'sname' löschen		re-	re-
delete(userid)		turn	turn
n	Fehler ?	j	turn
return(0)	return(-3)	(-4)	(-1) (-2)

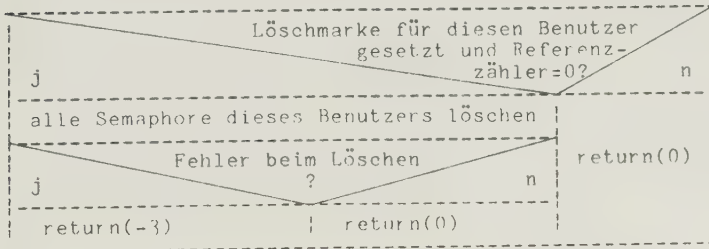
delsem(userid)

Parameter: userid - Benutzeridentifikation des Benutzers, für den alle Semaphore gelöscht werden sollen



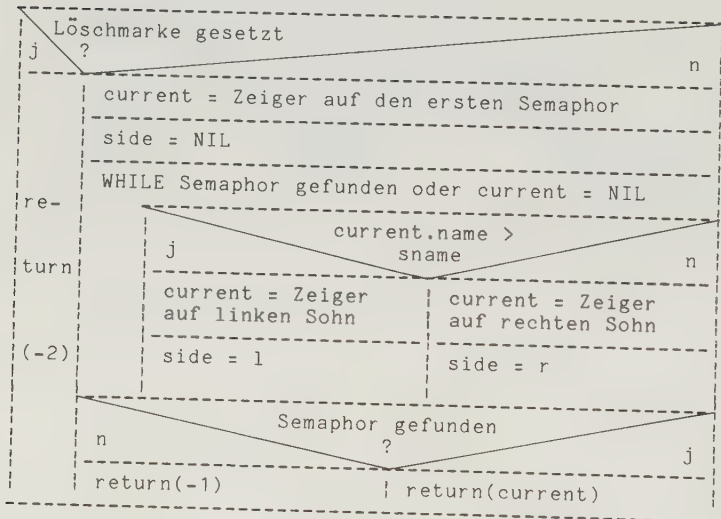
delete(userid)

Parameter: userid - Benutzeridentifikation



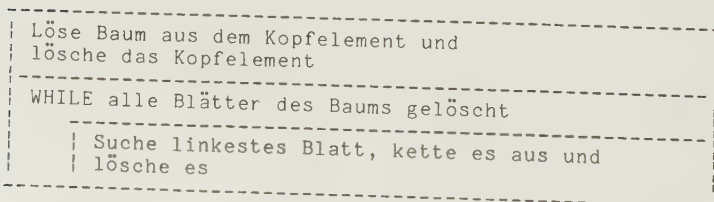
semasearch(sname,side)

Parameter: sname - Semaphorname
 Returnparameter: side - Verkettungshinweis
 =1 Semaphor ist linker Sohn
 =r Semaphor ist rechter Sohn
 =NIL Semaphor ist Kopf



semadel(pheader)

Parameter: pheader - Zeiger auf Kopfelement des Benutzers



5.6 Nachrichtenkopplungs- und Kommunikationssystem

5.6.1 Nachrichtenkopplungssystem - Hardware

Wie schon in Kapitel 3 kurz beschrieben, existiert neben der Speicherkopplung eine Nachrichtenkopplungsebene. Die Hardware für das Nachrichtensystem ist in Zusammenarbeit mit dem Lehrstuhl III des IMMD im Rahmen eines Seminars /Hess77/ entwickelt und im Labor des Instituts gebaut worden. Ein nahezu identisches System ist auch im EGPA-Projekt /Hess79/ realisiert.

Besondere Berücksichtigung bei der Entwicklung des Systems fand der Aspekt, den Ausfall von Hardwarekomponenten des Nachrichtensystems selbst und Ausfälle ganzer Teilsysteme in ihrer Auswirkung zu begrenzen. Stromversorgungszusammenbrüche einzelner Teilsysteme führen z.B. nicht zu einer Blockade des Nachrichtenbusses. Außerdem können über das Nachrichtensystem mit geeigneter Softwareunterstützung fehlerhaft arbeitende Teilsysteme lokalisiert und, falls nötig, isoliert werden.

Über die Standard-E/A-Hardwareschnittstelle ist das Nachrichtensystem von jedem Prozessor aus direkt erreichbar (siehe Abb.3.2). Jedem Prozessor ist ein Empfangs- und ein Sendeteil zugeordnet, die parallel betrieben werden können. Im Empfangsteil ist ein Ringpuffer realisiert, der bis zu sechzehn Nachrichtenblöcke zwischenspeichern kann. Das parallele Einschreiben und Lesen der Nachrichten ist möglich.

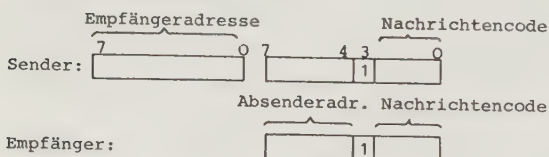
Um gleiche Nachrichten mehreren Prozessoren gleichzeitig schicken zu können, wird die Adressierung der Prozessoren

mit Hilfe eines 1-aus-8-Codes¹ durchgeführt. Eine Nachricht auf dem Nachrichtenbus wird immer von den Prozessoren übernommen, deren Adreßbits gesetzt sind. Mit der Übernahme einer Nachricht in seinen Nachrichtenpuffer sendet der Empfangsteil ein Unterbrechungssignal (Interrupt) an seinen Prozessor. Nicht vollständige (falsch formatierte) Nachrichten werden von der Hardware erkannt und gekennzeichnet.

5.6.2 Nachrichtenkopplungssystem - Software

Entsprechend den Hardwareeinheiten ist auch die Software des Nachrichtensystems in einen Sender- und einen Empfängermodul gegliedert. Die Gerätetreiber der Nachrichtensystemhardware entsprechen im Aufbau denen des E/A-Systems vom Typ 'unstrukturierte E/A'. Auf eine nähere Beschreibung kann deshalb hier verzichtet werden.

Nachrichtenformat 1:



Nachrichtenformat 2:

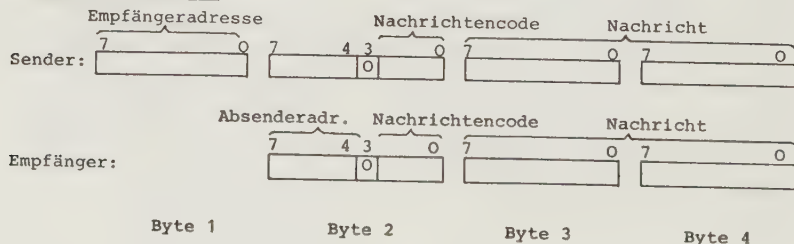


Abb. 5.6.1 Nachrichtenformate

1) auf 1-aus-16-Code erweiterbar

5.6.2.1 Sendermodul

Dem Kommunikationssystem (Kap. 5.6.3) stehen durch den Sendemodul folgende Funktionen zur Verfügung:

- 1) Initialisiere Sender
- 2) Sende Nachricht
- 3) Generiere Alarm

Die Nachrichten werden der Funktion 'Sende Nachricht' in einem Puffer, dessen Adresse als Parameter übergeben wird, zur Verfügung gestellt. Zulässig sind zwei Nachrichtenformate, deren Aufbau aus der Abb.5.6.1 hervorgeht.

Vor dem Absetzen einer Nachricht wird sie auf Korrektheit hinsichtlich Format, Gültigkeit der Zieladresse und Zulässigkeit des Nachrichtencodes überprüft.

Mit dem Aufruf der Funktion 'Generiere Alarm' wird eine festcodierte Nachricht an alle Prozessoren abgesetzt. Diese Funktion unterstützt die Implementierung eines Moduls, dessen Aufgabe es ist, Teilsystemausfälle bei Versorgungsspannungszusammenbrüchen in ihrer Auswirkung einzugrenzen (Kap. 5.6.3.5).

5.6.2.2 Empfängermodul

Das in Abb. 5.6.2 dargestellte Prozeßsystem zeigt die Einbettung des Empfängermoduls in die Empfängerseite des Kommunikationssystems. Eine standardisierte Schnittstelle, sowie ein eigenes Nachrichtenpuffersystem trennen den Nachrichteneingangsteil (Nachrichteneingangsprozeß) von dem Prozeßsystem, welches die aus den Nachrichten resultierenden Aktionen ausführt (Nachrichtenprozesse). Durch diese Trennung ist die leichte Erweiterbarkeit des Kommunikationssystems (Kap. 5.6.3) gewährleistet.

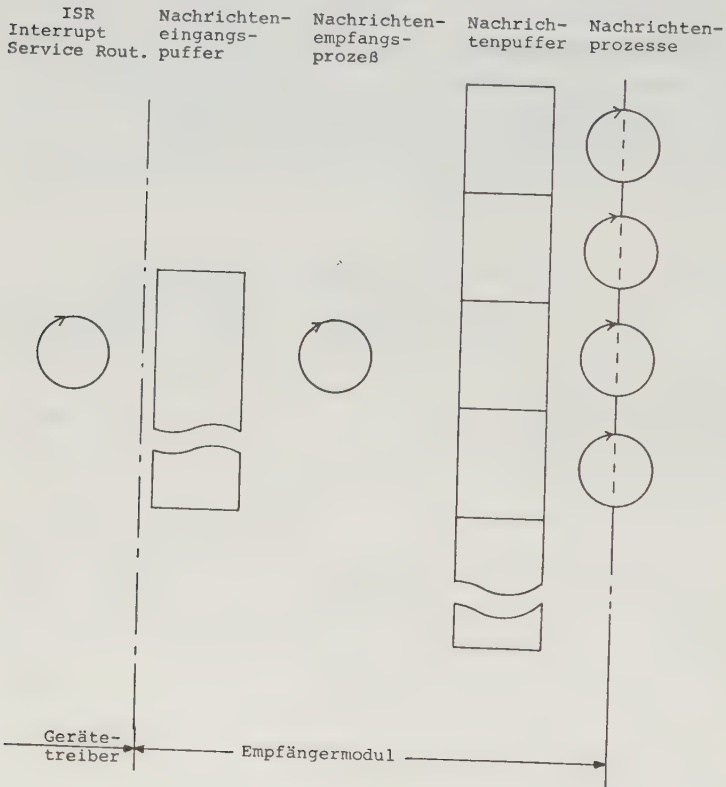


Abb. 5.6.2 Prozeßsystem des Kommunikationssystems
(Empfängerseite)

Für jeden Nachrichtentyp existiert ein Nachrichtenpuffer von nachrichtentypspezifischer Länge (Systemgenerierungsparameter) und mindestens ein Nachrichtenprozeß, dessen erste Aktion der Aufruf der Funktion 'Empfange Nachricht' ist. Diese Funktion übernimmt die notwendige Koordinierung mit

dem Nachrichtenempfangsprozess und stellt dem Aufrufer die für ihn bestimmte Nachricht zur Verfügung. Während der Initialisierung des Nachrichtensystems in der Anlaufphase des Gesamtsystems, werden alle Nachrichtenprozesse gestartet. Durch den Aufruf der Funktion 'Empfange Nachricht' blockieren sich die Prozesse an den leeren Nachrichtenpuffern mit einer der Nachrichtenankunftsrate bzw. der Nachrichtenreaktionsgeschwindigkeit entsprechenden Priorität. Analog wird mit dem Nachrichtenempfangsprozess verfahren, der in der Anlaufphase zusätzlich die Initialisierung der Empfängerhardware übernimmt.

Die Nachrichtenübertragung in den Nachrichteneingangspuffer sowie die Belegung des Nachrichtenbusses wird in der für das E/A-System üblichen Weise abgewickelt.

5.6.2.3 Spezifikation

MODULE Nachrichtenkopplungssystem / mess /

TYPES

```

    bufp:          DESIGNATOR;                                * Zeiger auf Nach-
                                                            richtenpuffer-
                                                            element

    messagecode: {messcode1,
                  messcode2,
                  .
                  .
                  .
                  messcodemax
                };                                           * Menge der Nach-
                                                            richtencodes

    messformat: {short,
                 long,
                 bad
                };                                           * Menge der Nach-
                                                            richtenformate

    message:       STRUCT(
                    messagecode mc,      * Nachrichtencode
                    messformat  format,  * Nachrichten-
                                         format
                    VEC_OF_INT  inf      * Nachricht
                );                                           * Nachrichten-
                                                            struktur

    typebuffer:    STRUCT(
                    BOOLEAN        blocked, * Zustand des Nach-
                                         richtenprozesses
                    bufp           nxtmess, * Zeiger auf die
                                         älteste Nachricht
                                         im Puffer

```

```
        bufp          nxtfree, * Zeiger auf den
                                ersten freien
                                Platz im Puffer

        VECTOR_OF
        message      mess      * Pufferbereich
                                der Nachrichten

    );                                * Nachrichten-
                                puffer
```

DECLARATIONS

```
typebuffer          messin;
VEC_OF_typebuffer   messbuf;
message             mess;
puaff               rec;
messagecode         messcode;
```

PARAMETERS

```
INTEGER            messinmax;      * Max. Länge des Nachrichten-
                                puffers

VEC_OF_INT         messbufmax;     * Max. Länge der nachrich-
                                tenspez. Nachrichtenpuffer

VEC_OF_INT         prio;           * Nachrichtenprspez. Prio.

INTEGER            messinprio;     * Prio. des Nachrichten-
                                empfangsprozesses
```

EXTERNALREFS

```
proc.m.puaff;
OFUN proc.sleep(chan,prio);
OFUN proc.wakeup(chan);
OFUN io.write(filename,↑buff);
```

FUNCTIONS

OVFUN receivemess(messcode) → mess;

* Parameter: messcode - Nachrichtencode

* Effekt: Übernimmt Nachrichten aus dem
Nachrichtenpuffer

EXCEPTIONS

messcode ↦ messagecode;

EFFECTS

procblock(messcode);

message = messbuf(messcode).nxtmess;

update(messbuff(messcode));

OFUN procblock(messagecode)

* Parameter: messagecode - Nachrichtencode

* Effekt: Falls Puffer leer, wird der
Nachrichtenprozess blockiert

HIDDEN;

EXCEPTIONS

none;

EFFECTS

TYPECASE messbuf(messmessagecode).nxtmess = UNDEFINED OF

TRUE: messbuf(messmessagecode).blocked = TRUE;

sleep(messbuf(messagecode),prio(messagecode));

FALSE: ;

END

OFUN getmes;

* Effekt: Übernimmt Nachricht aus dem
Nachrichtenpuffer, decodiert sie
und hinterlegt sie in den dem
Nachrichtencode entsprechenden
Puffer.

EXCEPTIONS

messin.nxtmess.messcode $\neq$ messagecode;
messbuf(messin.nxtmess.messcode).nxtfree = UNDEFINED;
messin.nxtmess.format = bad;

EFFECT

mc = messin.nxtmess.messcode;
messbuf(mc) = messin.nxtmess;
update(messin);
update(messbuf(mc));
proccont(messbuf(mc));

OFUN proccont(messagecode);

* Parameter: messagecode - Nachrichtencode
* Effekt: Falls Nachrichtenprozess blockiert
 → setze Prozess in 'RUN'-Status
 Falls Nachrichteneingangspuffer leer
 blockiere Nachrichtenempfangsprozess

HIDDEN;

EXCEPTIONS

none;

EFFECT

TYPECASE messbuf(messagecode).blocked OF
 TRUE: wakeup(messbuf(messagecode));
 messbuf(messagecode).blocked = FALSE;
 FALSE: ;
END
TYPECASE messin.nxtmess = UNDEFINED OF

```
TRUE: messin.blocked = TRUE;
      sleep(messin,messinprio);
FALSE: ;
END
```

OFUN update(buffer);

```
* Parameter: buffer - Pufferid.
* Effekt:      Implementierungsabhängige
               Anpassung der Pufferverwaltungs-
               informationen an den aktuellen
               Zustand
```

HIDDEN;

EFFECTS

not def.;

OFUN putmess(rec,messcode,↑mess);

```
* Parameter: rec - Zielprozessor
               messcode - Nachrichtencode
               ↑mess - Zeiger auf Nachrichten-
                       puffer
* Effekt:      Sendet die im spez. Puffer hinter-
               legte Nachricht an die in 'rec' ange-
               gebenen Prozessoren
```

EXCEPTIONS

```
messcode ↯ messagecode;
mess.format ↯ messformat;
rec ↯ puaff;
```

EFFECT

```
write(/dev/icimm,↑mess);
```

END_MODULE

5.6.3 Kommunikationssystem

Das auf dem Nachrichtenkopplungssystem aufbauende Kommunikationssystem ist nicht im bisher verwendeten Sinne ein eigener Modul, sondern die Zusammenfassung aller das Nachrichtensystem benutzenden Komponenten:

- 1) Koordinierungssystem der Systemprozesse
 - 'signal'-Operationen auf Fremdprozessoren
- 2) Koordinierungssystem der Benutzerprozesse
 - 'signal'-Operationen auf Fremdprozessoren
- 3) Prozeßverwaltung
 - Prozeß/Prozessorwechsel
- 4) Bussystemverwaltung
 - Busschalterkontrolle des gemeinsamen Selektorbusses
 - Busschalterkontrolle des gemeinsamen Multiplexorbusses
- 5) Koordinierungssystem der Systemprozesse
 - Lockoperationen ohne Speicherbelastung bei aktivem Warten
- 6) Alarmsystem
- 7) Test- und Überwachungssystem
 - Initialisiere und teste Nachrichten- und Kommunikationssystem

Diese Komponenten sind zum Teil bereits realisiert, zum Teil nur konzipiert.

5.6.3.1 'signal'-Operationen auf Fremdprozessoren

Ein für Polyprozessorsysteme typisches Problem ist die Tatsache, daß Prozesse, die eine 'signal'-Operation ausführen und Prozesse, die aufgrund dieser Operation in den lauffähigen Zustand überführt werden müssen, auf verschiedenen Prozessoren residieren können.

Als Lösungsmöglichkeiten bieten sich hier an:

1) Implementierung eines 'Polling'-Mechanismus

Vorteile : -Kein Nachrichtenkopplungssystem notwendig

Nachteile: -'Polling'-Zyklus müßte auf die größte zu erwartende Synchronisationsrate abgestimmt werden. Dies würde eine bedeutende Belastung des gemeinsamen Speichers bedeuten.

-zusätzliche Belastung der Zentraleinheit durch häufige Prozeßwechsel

2) Implementierung der Prozeßverwaltung oder mindestens der für die Problemlösung relevanten Teile als globale Moduln.

Vorteile : -kein Nachrichtenkopplungssystem notwendig

Nachteile: -Keine Verdrängungsstrategie möglich, es sei denn mit dem unter 1) vorgeschlagenen Mechanismus

-für das Wecken eines nicht aktiven Prozessors wäre ein 'Polling'-Mechanismus erforderlich.

-würde eine völlige Neufassung der Prozeßverwaltung des UNIX - Betriebssystems erforderlich machen

3) Implementierung eines Nachrichtenmechanismus

Vorteile : -Nachteile der Lösungen 1) - 2) entfallen.

Nachteile: -Nachrichtenkopplungssystem notwendig

Die positiven Eigenschaften der Alternative 3) machen die Vorteile einer zusätzlichen Nachrichtenkopplung deutlich.

Die Lösung wurde in den Moduln 'syssem' bzw. 'sema' (Kap. 5.5) implementiert.

5.6.3.2 Prozeß/Prozessorwechsel

Im Kapitel 5.2.1.4 ist ausführlich der Prozessorwechsel eines Prozesses diskutiert worden. Ähnlich wie für die 'signal'-Operationen besteht auch hier die Alternative des 'Polling'-Mechanismus mit den oben beschriebenen Nachteilen.

Den Nachrichtenmechanismus in der Prozeßverwaltung realisieren die Funktionen 'setintm' und 'setintmcall', die im Kapitel 5.2.2, letztere als interne Funktion ('switch'), spezifiziert sind.

5.6.3.3 Bussystemverwaltung

Wie aus der Hardwarebeschreibung in Kapitel 3 hervorgeht, werden die gemeinsamen Busse mit Hilfe von Busschaltern, die durch E/A-Befehle steuerbar sind, den einzelnen Prozessen zugeordnet. Um ein häufiges Umschalten der gemeinsamen Busse zu vermeiden, sollte ein bestimmtes Teilsystem vorzugsweise den E/A-Verkehr abwickeln. Notwendig ist hier eine aufwendige Kommunikation auf Gerätetreiberebene mit speziellen Lösungen für den Selektorkanalbus und den Multiplexorbus. Bei Zugriffen auf gemeinsame Dateisysteme müßte neben der globalen Verwaltung dieser Dateisysteme ein Mechanismus installiert werden, der trotz des 'Cachesystems' der strukturierten E/A (Kap. 2.1), auch bei wechselnden Buszuordnungen die Konsistenz der Plattendaten garantiert.

5.6.3.4 Lockoperationen ohne Speicherbelastung bei aktivem Warten

Bei der im Kap.5.5 vorgeschlagenen Lockoperation handelt es sich um sogenannte 'Spin Lock'-Operationen, Lockoperationen, die in einer Schleife ständig ($10\mu\text{s}$ -Zyklus) die Lockvariable abfragen, falls sie gesetzt ist. Die ständigen Abfragen belasten den gemeinsamen Speicherbus und behindern damit die Prozesse, die auf den gemeinsamen Speicher Operationen ausführen wollen. Denkbare Abhilfen wären:

- 1) den Abfragezyklus durch eingeschobene Befehle ohne Zugriff auf den gemeinsamen Speicher zu vergrößern oder
- 2) den Prozeß ohne Prozessorabgabe zu blockieren

Die zweite Lösung ließe sich analog der 'Kernel Locks' im Betriebssystem HYDRA /Jone80/ mit Hilfe des Nachrichtensystems lösen.

Da der Aufwand der Implementierung unter Verwendung des vorliegenden Nachrichtensystems relativ gering ist, soll zu Studienzwecken eine Realisierung dieser Operation erfolgen. Messungen sollen zeigen, ob diese Lösung bei der gegebenen Hardware Vorteile gegenüber den 'Spin Lock'-Operationen erbringen kann.

Vermutlich ist das unter 2) beschriebene Verfahren nur dann sinnvoll, wenn man sich auf eine direkte 'Hardwarekoordination' abstützen kann und damit hinreichend schnell ist.

5.6.3.5 Alarmsystem

Die Zuverlässigkeit von Polyprozessorsystemen der in dieser Arbeit betrachteten Klasse läßt sich nur dann steigern, wenn es gelingt, Ausfälle oder Fehlverhalten von Teilsystemen in ihren Auswirkungen genau auf diese Teilsysteme zu beschränken.

Die Chance einer Lösung dieses Problems besteht dann, wenn bei einem Totalausfall Prozessorstatus und Prozeßstatus des zum Zeitpunkt des Ausfalls laufenden Prozesses verfügbar sind. Ferner muß bekannt sein, welche Koordinierungsvariablen der Prozeß zum Ausfallzeitpunkt belegt hatte.

Bei den hier verwendeten Prozessoren wird bei Zusammenbruch der Versorgungsspannung automatisch der Prozessorstatus in den zugehörigen Prozessorkontrollblock im gemeinsamen Speicher gerettet. Zusätzlich wird eine bestimmte (angebbare) Adresse angesprungen und dort hinterlegter Code, dessen Ausführungszeit 1 ms nicht überschreiten darf, ausgeführt. Diese Zeit genügt mit Sicherheit, eine Alarmnachricht an das Nachrichtensystem abzusetzen.

Von den empfangenden Prozessoren fällt dem ersten reagierenden die Aufgabe zu, das ausgefallene Teilsystem aus dem Gesamtsystem auf Betriebssystemebene herauszulösen. Hiervon sind die Module Prozessorverwaltung, Speicherverwaltung, Nachrichtensystem und Synchronisationssystem betroffen. Je nach Prozeßklassenzugehörigkeit ist es möglich, die auf den ausgefallenen Prozessor residierenden Prozesse auf einen anderen Prozessor fortzusetzen (Prozesse der Klasse 'global') oder aber erneut zu starten (Prozesse der Klasse lokal, global/temp.lokal). Für den letzteren Fall ist die Existenz des im Kapitel 5.6.3.3 beschriebenen Moduls 'Bussystemverwaltung' Voraussetzung.

Eine Meldung auf der Systemkonsole wird das ausgefallene Teilsystem und die davon betroffenen Prozesse protokollieren.

5.6.3.6 Test- und Überwachungssystem

In diesem Modul sind die für die Anlaufphase notwendigen Funktionen zur Initialisierung des Nachrichten- und Kommunikationssystems implementiert. Weiter sollen in ihm Komponenten integriert werden, mit deren Hilfe Aussagen über die Funktionstüchtigkeit des Gesamtsystems möglich sein sollen.

Als erster Ansatz ist ein Test des Nachrichtensystems konzipiert, der in der Anlaufphase feststellen soll, welche Komponenten des Nachrichtensystems und daraus folgernd welche Teilsysteme über das Nachrichtensystem ansprechbar sind. In diesem Test sendet jeder Prozessor "n" an seinen Nachbarn " $n+1 \bmod \text{Prozessoranzahl}$ " eine Nachricht, die dieser quittiert, indem er allen Prozessoren (ausgenommen sich selbst) die empfangene Nachricht sendet. Nach Ablauf dieser Prozedur muß jeder Prozessor von jedem eine Nachricht erhalten haben. Nichterhaltene Nachrichten werden über Meldungen auf der Konsole protokolliert und durch eine Fehlerdiagnose ergänzt. Durch entsprechende Masken wird dafür Sorge getragen, daß im weiteren Betrieb keine Nachrichten an nicht betriebsbereite Teilsysteme gesendet werden.

Die beschriebene Vorgehensweise erlaubt in Dreiergruppen die eindeutige Bestimmung fehlerhafter Einheiten, wenn insgesamt mindestens zwei Einheiten korrekt arbeiten. Unter diesen Voraussetzungen können folgende Fehler erkannt werden:

- Teilsystem x sendet, empfängt aber nicht
- Teilsystem x empfängt, sendet aber nicht
- Teilsystem x empfängt und sendet nicht.

6.0 Betriebssystemschnittstelle, Binder und System- dienstbibliothek

6.1 Betriebssystemschnittstelle

Wie gefordert wurde die Betriebssystemschnittstelle des Monoprozessorsystems /Thom75/ beibehalten, d.h. also, daß alle auf einem Monoprozessorsystem ablauffähigen Programmsysteme ohne jede Einschränkung auch auf dem Polyprozessorsystem ablauffähig sind. Das System verhält sich dieser Gruppe der Anwender gegenüber wie ein Monoprozessorsystem.

Um die vorhandene Hardwarestruktur dem Benutzer für die Lösung seiner Probleme zugänglich zu machen, werden ihm zusätzliche Betriebssystemdienste angeboten, die die Implementierung paralleler Prozeßsysteme auf Benutzerebene gestatten.

Es sind dies die Dienste

zur Prozeßsteuerung:

`pfork(procaff,ptyp,res)`

Effekt: Erzeugen eines Prozesses der Klasse 'ptyp' auf einem Prozessor gemäß der angegebenen Prozessoraffinität (1-aus-8-Code).

`procaff = 0`: beliebiger Prozessor

`res = true`: Prozeß wird nicht ausgelagert

Returnpar: -1 - Fehler

0 - Returnparameter an den erzeugten Prozeß

>0 - Prozeßidentifikator des erzeugten Prozesses als Returnparameter an den erzeugenden Prozeß

pjoin(pn₁,pn₂,....pn_{max},tout)

Effekt: Warten auf das Terminieren der Prozesse pn₁,
pn₂,...pn_{max}

Returnpar: -1 - Prozeß nicht bekannt
-2 - Zeitkonstante überschritten
0 - Prozesse sind terminiert

pstop(pn₁,pn₂,....pn_{max})

Effekt: Stoppen der Prozesse pn₁,pn₂,....pn_{max} ohne
Freigabe des Speichers

Returnpar: >1 - Mindestens Prozeß 'Returnpar.' nicht
bekannt
0 - Prozesse sind gestoppt

pstart(pn₁,pn₂,....pn_{max})

Effekt: Wiederaufsetzen der Prozesse pn₁,pn₂,....pn_{max}
(Restart). Die Prozesse mußten vorher durch
den Betriebssystemdienst 'pstop' gestoppt wor-
den sein.

Returnpar: >1 - Mindestens Prozeß 'Returnpar.' nicht
im 'STOP'-Zustand
0 - Prozesse gestartet

Die Betriebssystemdienste 'pstop' und 'pstart' sind noch
nicht implementiert.

zur Prozeßkoordinierung

crsem(sname,initval,tout)

Effekt: Erzeugt einen Semaphor mit dem Namen 'sname'
und dem Anfangswert 'initval'. Die Zeitüber-
wachungskonstante dient der Auflösung von Ver-
klemmungen durch fehlerhafte Koordinierung.

Returnpar: <0 - Fehler Beim Erzeugen des Semaphore

dstsem(sname)

Effekt: Löschen des Semaphore 'sname'

Returnpar: <0 - Fehler beim Löschen des Semaphore

delsem(userid)

Effekt: Anstoßen der Löschoperation für alle Semaphore, die für den Benutzer 'userid' eingerichtet sind.

Returnpar: <0 - Fehler beim Löschen der Semaphore

vsem(sname,ch)

Effekt: 'signal'-Operation mit 'Chunk'-Wert 'ch'

Returnpar: <0 - Fehler - Semaphor nicht für 'userid' definiert

psem(sname,ch)

Effekt: 'wait'-Operation mit 'Chunk'-Wert 'ch'

Returnpar: <0 - Fehler - Semaphor nicht für 'userid' definiert

Mit den obigen Prozeßkoordinierungssystemdiensten steht dem Benutzer ein Koordinierungssystem des Typs PV_{chunk} zur Verfügung.

Da eine Koordinierungsoperation über einen Betriebssystemdienst in der Größenordnung 1-5ms dauert (Abschätzung), ist daran gedacht, dem Benutzer auch eine Lockoperation als Systemdienst anzubieten. In jedem Fall ist vom Benutzer auf ein vernünftiges Verhältnis Koordinierungszeit/Parallellaufabschnitte ohne Koordinierungsoperationen, zu achten.

6.2 Binder, Systemdienstbibliothek

Bei der Lösung, Daten gemäß ihrer Attribute 'lokal' oder 'global' in separate Segmente abzulegen, ist (in der vorläufigen Version) die Eigenschaft des 'C'-Compilers genutzt worden, globale initialisierte Daten zusammenhängend in der Objektdatei abzulegen.

Dem modifizierten Binder wird durch die Angabe eines Parameters angezeigt, ob in dem Lademodul separate Datensegmente oder ein zusammenhängendes Datensegment angelegt werden soll. Entsprechend wird der Laderkontrollblock aufbereitet.

Neben der Erweiterung der Systemdienstbibliothek um die neu eingeführten Systemdienste (Kap. 6.1) mußte die gesamte Bibliothek hinsichtlich der globalen Daten überarbeitet werden. Die Systemdienstaufrufe sind jetzt so abgefaßt, daß sie für alle Prozeßklassen, soweit sinnvoll, gleichermaßen anwendbar sind. Weitere Informationen über den Binder sind den UNIX-Handbüchern /Thom75/ zu entnehmen.

sollte in einer separaten Datei geschehen, die durch die Compileranweisung '# include' bei der Übersetzung der verschiedenen Quelldateien berücksichtigt wird. Hierdurch wird sichergestellt, daß für alle Prozesse des Prozeßsystems die gemeinsamen Daten die gleiche Struktur besitzen, eine Notwendigkeit, die weder der Compiler noch der Binder zu überprüfen in der Lage sind.

Für jeden Prozeß wird mit Hilfe des Binders ein separater Lademodul erzeugt. Zur Zeit (bis zur Fertigstellung des Moduls 'Bussystemverwaltung' Kap. 5.5.3.3) muß der Benutzer selbst dafür Sorge tragen, daß bei der Ausführung des Betriebssystemdienstes 'exec' die zu ladende Datei physikalisch von dem entsprechenden Prozessor erreichbar ist.

In Abb.7.1 ist die Struktur eines möglichen Prozeßsystems paralleler Prozesse mit einigen der zur Verfügung stehenden Prozeßsteueranweisungen dargestellt.

7.1 Ein Beispiel

In diesem Abschnitt ist als Beispiel für ein Benutzersystem paralleler Prozesse das bekannte Philosophenproblem /Dijk71/, in einer Lösung von Hofmann /Hofm74/, ausgewählt worden.

In der Datei 'phil.h' sind die prozeßsystemglobalen Daten, hier die Konstanten und der Zustandsvektor der Philosophen, deklariert. In der Datei 'philgen.c' ist der Steuerprozeß formuliert, der die Semaphoren anlegt und initialisiert und die 'Philosophenprozesse' (Dateien: 'phil0.c' - 'phil4.c') erzeugt. Durch die Konstante 'ZYKLUS' ist dafür gesorgt, daß die 'Philosophenprozesse' nach einer gewissen Zeit terminieren. Der Steuerprozeß wartet auf dieses Ereignis mit der Funktion 'pjoin'. Tritt dieses Ereignis nicht innerhalb der Zeit 'TOUTJOIN' ein, so werden die 'Philosophenprozesse' durch den Betriebssystemdienst 'kill' aus dem System ent-

fernt. Vor dem Terminieren des Steuerungsprozesses werden mit der Funktion 'delsem' sämtliche erzeugte Semaphoren des Prozeßsystems freigegeben.

Aus Gründen der Übersichtlichkeit ist in diesem Beispiel auf eine Prüfung der Fehlercodes in den Rückgabeparametern bis auf den oben beschriebenen Fall 'pjoin', verzichtet worden.

```
/* */
/* Prozesssystemglobale Vereinbarungen (phil.h) */
/* */
/* Code: Prozessklasse global/temp. lokal */

#define gtl 1

/* Code: Setze Prozess resident */

#define res 1

/* Prozessoraffinitaet: 1 aus 8 - Code */

#define PROC1 1
#define PROC2 2
#define PROC3 4

/* Semaphornamen */

#define EX 0
#define PHILO 1
#define PHIL1 2
#define PHIL2 3
#define PHIL3 4
#define PHIL4 5

/* Timeout - Konstanten */

#define TOUTSEM 5
#define TOUTJOIN 200

#define DENKEN 100000
#define ESSEN 100000
#define ZYKLUS 20

/* Zustandsvektor der Philosophen */

int c[] {0,0,0,0,0};

/* Ladeobjektdateinamen der Philosophen */

char *fphil0 "/phil0.out ";
char *fphil1 "/phil1.out ";
char *fphil2 "/phil2.out ";
char *fphil3 "/phil3.out ";
char *fphil4 "/phil4.out ";
```

```
/*
/*      Init. Philosophen (philgen.c)      */
/*
/*
#include "phil.h"
main()
{
    int i,id0,id1,id2,id3,id4;
    int uid;
    int argumente[11];
    int *argm;

    i = crsem(EX,1,TOUTSEM);
    i = crsem(PHIL0,0,TOUTSEM);
    i = crsem(PHIL1,0,TOUTSEM);
    i = crsem(PHIL2,0,TOUTSEM);
    i = crsem(PHIL3,0,TOUTSEM);
    i = crsem(PHIL4,0,TOUTSEM);

    id0 = pfork(PROC1,gtl,res);
    if(id0 == 0)
        /* phil0 */
        i = execl(fphil0,argm);

    id1 = pfork(PROC2,gtl,res);
    if(id1 == 0)
        /* phil1 */
        i = execl(fphil1,argm);

    id2 = pfork(PROC3,gtl,res);
    if(id2 == 0)
        /* phil2 */
        i = execl(fphil2,argm);

    id3 = pfork(PROC1,gtl,res);
    if(id3 == 0)
        /* phil3 */
        i = execl(fphil3,argm);

    id4 = pfork(PROC2,gtl,res);
    if(id4 == 0)
        /* phil4 */
        i = execl(fphil4,argm);

    i = pjoin(id0,id1,id2,id3,id4,TOUTJOIN);
    if(i == -2)
        {kill(id0,9);
         kill(id1,9);
         kill(id2,9);
         kill(id3,9);
         kill(id4,9); }

    uid = getuid();
    i = delsem(uid);
    i = exit();
}
```

```
/*                                     */
/*      Philosoph 0 (phil0.c)         */
/*                                     */

#include "phil.h"

main()
{int i, j;

  for(j = 0; j < ZYKLUS; j++)
  {
    for(i = 0; i < DENKEN; i++);
    psem(EX,1);
    c[0] = 1;
    test (0);
    vsem(EX,1);
    psem(PHILO,1);
    for (i = 0; i < ESSEN; i++);
    psem(EX,1);
    c[0] = 0;
    test (4);
    test (1);
    erbarmen (4);
    erbarmen (1);
    vsem (EX,1);
  }
  i = exit();
}
test (i)
int i;
{int j;
  if (((c[i-1]%5] < 2) && (c[i] = 1) && (c[(i+1)%5] < 2))
      || (c[i] = 2))
  {c[i] = 3;
    switch(i)
    {case 0: { vsem(PHILO,1);
               break; }
      case 1: { vsem(PHIL1,1);
               break; }
      case 2: { vsem(PHIL2,1);
               break; }
      case 3: { vsem(PHIL3,1);
               break; }
      case 4: { vsem(PHIL4,1);
               break; }
    }
  }
}
erbarmen (i)
int i;
{if (c[i] == 1 && (c[(i-1)%5] == 3 || c[(i+1)%5] == 3))
    c[i] = 2;
}
```

```
/*
/*      Philosoph 1 (phil1.c)      */
/*
/*
#include "phil.h"

main()
{int i, j;

  for(j = 0; j < ZYKLUS; j++)
  {
    for(i = 0; i < DENKEN; i++);
    psem(EX,1);
    c[1] = 1;
    test (1);
    vsem(EX,1);
    psem(PHIL1,1);
    for (i = 0; i < ESSEN; i++);
    psem(EX,1);
    c[1] = 0;
    test (0);
    test (2);
    erbarmen (0);
    erbarmen (2);
    vsem (EX,1);
  }
  i = exit();
}
test (i)
int i;
{int j;
  if ((c[i-1%5] < 2) && (c[i] = 1) && (c[(i+1)%5] < 2))
    || (c[i] = 2))

    {c[i] = 3;

      switch(i)
      {case 0: { vsem(PHIL0,1);
                  break; }
        case 1: { vsem(PHIL1,1);
                  break; }
        case 2: { vsem(PHIL2,1);
                  break; }
        case 3: { vsem(PHIL3,1);
                  break; }
        case 4: { vsem(PHIL4,1);
                  break; }
      }
    }
}
erbarmen (i)
int i;
{if (c[i] == 1 && (c[(i-1)%5] == 3 || c[(i+1)%5] == 3))
    c[i] = 2;
}
```

```

/*
/*      Philosoph 2 (phil2.c)      */
/*
/*
#include "phil.h"

main()
{int i, j;

  for(j = 0; j < ZYKLUS; j++)
  {
    for(i = 0; i < DENKEN; i++);
    psem(EX,1);
    c[2] = 1;
    test (2);
    vsem(EX,1);
    psem(PHIL2,1);
    for (i = 0; i < ESSEN; i++);
    psem(EX,1);
    c[2] = 0;
    test (1);
    test (3);
    erbarmen (1);
    erbarmen (3);
    vsem (EX,1);
  }
  i = exit();
}
test (i)
int i;
{int j;
  if (((c[i-1%5] < 2) && (c[i] = 1) && (c[(i+1)%5] < 2))
      || (c[i] = 2))

    {c[i] = 3;

      switch(i)
      {case 0: { vsem(PHILO,1);
                  break; }
        case 1: { vsem(PHIL1,1);
                  break; }
        case 2: { vsem(PHIL2,1);
                  break; }
        case 3: { vsem(PHIL3,1);
                  break; }
        case 4: { vsem(PHIL4,1);
                  break; }
      }
    }
}
erbarmen (i)
int i;
{if (c[i] == 1 && (c[(i-1)%5] == 3 || c[(i+1)%5] == 3))
    c[i] = 2;
}

```

```
/*
/*      Philosoph 3 (phil3.c)
/*
/*

#include "phil.h"

main()
{int i, j;

  for(j = 0; j < ZYKLUS; j++)
  {
    for(i = 0; i < DENKEN; i++);
    psem(EX,1);
    c[3] = 1;
    test (3);
    vsem(EX,1);
    psem(PHIL3,1);
    for (i = 0; i < ESSEN; i++);
    psem(EX,1);
    c[3] = 0;
    test (2);
    test (4);
    erbarmen (2);
    erbarmen (4);
    vsem (EX,1);
  }
  i = exit();
}
test (i)
int i;
{int j;
  if ((c[i-1%5] < 2) && (c[i] = 1) && (c[(i+1)%5] < 2))
    || (c[i] = 2))

    {c[i] = 3;

      switch(i)
      {case 0: { vsem(PHIL0,1);
                  break; }
        case 1: { vsem(PHIL1,1);
                  break; }
        case 2: { vsem(PHIL2,1);
                  break; }
        case 3: { vsem(PHIL3,1);
                  break; }
        case 4: { vsem(PHIL4,1);
                  break; }
      }
    }
}
erbarmen (i)
int i;
{if (c[i] == 1 && (c[(i-1)%5] == 3 || c[(i+1)%5] == 3))
    c[i] = 2;
}
```

```
/*                                                    */
/*      Philosoph 4 (phil4.c)                        */
/*                                                    */

#include "phil.h"

main()
{int i, j;

  for(j = 0; j < ZYKLUS; j++)
  {
    for(i = 0; i < DENKEN; i++);
    psem(EX,1);
    c[4] = 1;
    test (4);
    vsem(EX,1);
    psem(PHIL4,1);
    for (i = 0; i < ESSEN; i++);
    psem(EX,1);
    c[4] = 0;
    test (3);
    test (0);
    erbarmen (3);
    erbarmen (0);
    vsem (EX,1);
  }
  i = exit();
}
test (i)
int i;
{int j;
  if (((c[i-1%5] < 2) && (c[i] = 1) && (c[(i+1)%5] < 2))
      || (c[i] = 2))

    {c[i] = 3;

      switch(i)
      {case 0: { vsem(PHIL0,1);
                  break; }
        case 1: { vsem(PHIL1,1);
                  break; }
        case 2: { vsem(PHIL2,1);
                  break; }
        case 3: { vsem(PHIL3,1);
                  break; }
        case 4: { vsem(PHIL4,1);
                  break; }
      }
    }
  }
  erbarmen (i)
  int i;
  {if (c[i] == 1 && (c[(i-1)%5] == 3 || c[(i+1)%5] == 3))
      c[i] = 2;
  }
}
```

8.0 Schlußbemerkungen

In der vorliegenden Arbeit ist ein Weg aufgezeigt worden, mit den an der Hochschule zur Verfügung stehenden Mitteln Betriebssysteme für Polyprozessorsysteme zu entwerfen und zu implementieren.

Besonderer Wert wurde auf die Lösung des Problems gelegt, nur den Benutzern die Hardwarestruktur des Systems transparent zu machen, die für spezielle Aufgabenstellungen diese Struktur zu nutzen wünschen. Für den 'Normalbenutzer' verhält sich das Polyprozessorsystem wie ein Monoprozessorsystem, jedoch mit höherem Durchsatz und höherer Verfügbarkeit.

Durch die Einführung des Konzepts der lokalen Teilsysteme und durch die Realisierung des Nachrichten- und Kommunikationssystems ist es gelungen, den potentiellen Systemengpaß, den gemeinsamen Speicher, zu entlasten. Die hohe Modularisierung und die Trennung von "Strategie" und "Mechanismus" sind Voraussetzungen zur Erfüllung der Forderung, spezielle, aufgabenangepaßte Teilsysteme generieren zu können. Als erstes spezielles Teilsystem ist eine Variante ohne Hintergrundspeicherverwaltung vorgesehen.

Ein Problem, das in der vorliegenden Arbeit bisher nicht behandelt wurde, das jedoch mit der steigenden Komplexität der Softwaresysteme weiter an Aktualität gewinnen wird, ist das der Fehlersuche in parallelen Prozeßsystemen. Dieses trifft besonders auf Fehler der Kategorie zu, deren Auftreten von der gegenseitigen Verzahnung der beteiligten Prozesse abhängig ist und die sich erst mit einer zeitlichen Verzögerung äußern. Klassische Methoden, wie z.B. 'Tracesysteme' versagen hier, weil sie das Verhalten der Prozeßsysteme verändern; das kann soweit gehen, daß durch das 'Tracesystem' eine Zwangssynchronisation eintritt. Eine

strikte modulare Programmierung verbunden mit einem leistungsfähigen "feinen" Schutzsystem könnte die Einkreisung derartiger Fehler erleichtern.

Das entstehende Polyprozessorbetriebssystem wird ein Experimentiersystem sein, das insbesondere der praktischen Ausbildung der Studenten dienen soll. Dies wird dadurch besonders begünstigt, daß durch die hohe Modularität des Systems bedingt, Systemkomponenten in einer vertretbaren Zeit ausgetauscht oder modifiziert werden können. Außerdem wird dies durch die Tatsache unterstützt, daß 90% des Gesamtsystems in einer höheren Programmiersprache implementiert sind.

Literaturverzeichnis

- Bell78 The Bell System Technical Journal
 "UNIX - Time-Sharing-System"
 Vol. 57, No.6, Part 2 Juli/Aug 78 Bell Laboratories
 600 Mountain Ave., Murray Hill, N.J. 07974
- Bour78 S.R.Bourne
 "UNIX-Time Sharing System: The UNIX Shell"
 The Bell System Technical Journal, Vol.57, No.6,
 Part 2, Jul/Aug 78, Bell Lab. 600 Mountain Ave.,
 Murray Hill, N.J. 07974
- Denn76 Peter J. Denning
 "Fault Tolerant Operating-Systems"
 Computing Surveys, Vol.8, No.4, Dez.1976
- Dijk68 E.W. Dijkstra
 "Cooperating Sequential Processes"
 Programming Languages, ed F. Genuys, New York,
 Academic Press, 1968
- Feie71 R.J. Feiertag and E.I. Organick
 "The Multics Input-Output-System"
 Proc. Third Symposium on Operating System Prin-
 ciples, Okt. 71
- Hans77 Per Brinch Hansen
 "The Architecture of Concurrent Programs"
 Prentice-Hall, Inc. Englewood Cliffs, N.J. 07632,
 1977
- Hess77 H. Hessenauer, R. Klar
 "Entwurf von Hardwarekomponenten digitaler Rechner"
 Projektseminar WS 77/78 Univ. Erlangen-Nürnberg

- Hess79 H. Hessenauer
"EGPA-Projekt-Berichte"
IMMD - 1979
- Hofm74 F. Hofmann
"Betriebsprogrammierung I und II"
Vorlesung WS 74/75 und SS 75
- Hofm77 F. Hofmann
"Betriebsprogrammierung I und II"
Vorlesung WS 77/78 und SS 78
- Inte76 Perkin Elmer Computer Systems Division
"OS/32-Dokumentation Package"
Perkin Elmer Computer System Division, Oceanport,
N.J. 07757, Juni 76
- Inte79 Perkin Elmer Computer Systems Division
"OS/32-Dokumentation Package"
Perkin Elmer Computer System Division, Oceanport,
N.J. 07757, Febr. 79
- Jäpe77 Dieter Jäpel
"Ein Begriffssystem zur Formulierung von Schutzas-
pekten und seine Anwendung auf Programmsysteme"
Arbeitsberichte des IMMD Band 10 Nr. 7 1977
- Jone73 Anita K. Jones
"Protection in programmed systems"
Ph.D. Thesis Carnegie Mellon University Pittsburgh
Pa. Department of Computer Science 1973
- Jone80 Anita K. Jones and Peter Schwarz
"Experience Using Multiprocessor Systems -
A Status Report"
Computing Surveys, Vol.12, No.2, Juni 80

- Kera79 S. Keramidis and L. Mackert
"Specification and Implementation of Parallel Activities on Abstract Objects"
4th International Conference on Software Engineering Technical University of Munich, Sept. 79
- Kern78 W. Kernighan and D.M. Ritchie
"The C Programming Language"
Prentice-Hall, Inc. Englewood Cliffs, N.J. 07632, 1978
- Mill78 R. Miller
"UNIX - A Portable Operating System?"
Australian Universities Computing Science Seminar, Feb. 78
- Neal77 David Neal, Barbara North
"SOLO - Tutorials"
Kansas State University Manhattan, Kansas, Okt. 77
- Ritc78a D.M. Ritchie, S.C. Johnson, M.E. Lesk and B.W. Kernighan
"UNIX - Time Sharing System: The C Programming Language"
The Bell System Technical Journal, Vol.57, No.6, Part 2, Jul/Aug 78, Bell Lab. 600 Mountain Ave., Murray Hill, N.J. 07974
- Ritc78b D.M. Ritchie
"UNIX - Time Sharing System: A Retrospective"
The Bell System Technical Journal, Vol.57, No.6, Part 2, Jul/Aug 78, Bell Lab., 600 Mountain Ave., Murray Hill, N.J. 07974
- Ritc78c D.M. Ritchie and K. Thompson
"The UNIX Time Sharing System"
The Bell System Technical Journal, Vol.57, No.6, Part 2, Jul/Aug 78, Bell Lab., 600 Mountain Ave., Murray Hill, N.J. 07974

- Robi77 L. Robinson and O. Rouline
"SPECIAL - A Specification and Assertion Language"
Technical Report, CSL-46 Stanford Research Inst.,
Juni 77
- Salt74 Jerome H. Saltzer
"Ongoing Research and Development on Information
Protection"
ACM Operating System Review 8/3 Juli 1974
- Srod78 Ronald J. Srodawa
"Positive Experiences with a Multiprocessing Sys-
tem"
Computing Surveys Vol.10, No.1, März 1978
- Thom75 K. Thompson, D.M. Ritchie
"UNIX Programmer's Manual Sixth Edition"
Bell. Lab., 600 Mountain Ave., Murray Hill,
N.J.07974
- Thom78 K. Thompson
"UNIX Time Sharing System: UNIX Implementation"
The Bell System Technical Journal, Vol.57, No.6,
Part 2, Jul/Aug 78, Bell Lab., 600 Mountain Ave.,
Murray Hill, N.J. 07974
- Wend78 K.D. Wendler
"Betriebssystemaspekte in hierarchisch modularen
Polyprozessorsystemen - Modellierungsansätze und
Koordinierungsmechanismen"
Arbeitsberichte des IMMD, Univ. Erlangen-Nürnberg,
Band 11 No. 15, Okt. 78
- Wulf74 A.W. Wulf, E. Cohe, W. Corwin, A. Jones, R. Levin,
C. Pierson and F. Pollack
"HYDRA: The kernel of a multiprocessor operating
system"
Commun. ACM 17, 6(June 1974), 337-345

Lebenslauf

Claus-Uwe Linster

geb. am : 23.08.1941

in : Westerland/Sylt

Staatsang. : deutsch

Eltern : Aloys Linster
Ehefrau Magda, geb. Wegemann

Familienstand : verheiratet seit 25.09.1970
Ehefrau Karin, geb. Behnke
ein Kind

Schulen : 1948 - 1953 Volksschule Westerland/Sylt
1953 - 1959 Mittelschule Westerland/Sylt

Ausbildung : 1959 - 1960 Seemännische Grundausbildung
Seefahrtsschule Priwall/Lübeck
Reederei Bruns & Co Hamburg
1960 - 1964 Radio- und Fernsehtechnikerlehre
Fa. Schröter Westerland/Sylt
1965 - 1966 Techn. Praktikum
Fa. Siemens A.G. Berlin

Studium : 1966 - 1969 Staatl. Ingenieurakademie Gauss
Berlin, Fachrichtung: Elektronik
1969 Allgemeine Hochschulreife
1969 - 1973 Technische Universität Berlin
Fachrichtung: Elektrotechnik

Prüfungen : 1969 Staatl. Ingenieurprüfung
1973 Diplomprüfung - Elektrotechnik

Stellungen : seit 1973 Wiss. Angestellter am IMMD
Inf. IV der Univ. Erlangen - Nürnberg

